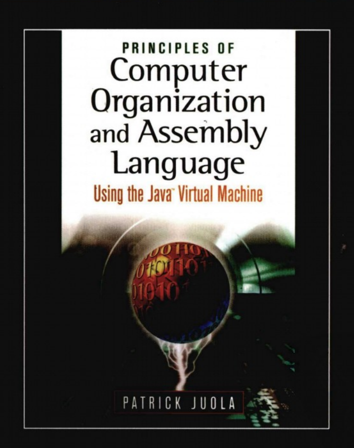
HZ BOOKS
华章教育

HALL

计 算 机 科 学 丛 书

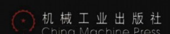
计算机组成及汇编语言原理

(美) Patrick Juola 迪尤肯大学 著 吴为民 艾丽华 张大伟 译



Principles of Computer Organization and Assembly Language

Using the Java Virtual Machine





计算机组成及汇编语言原理

本书以创新的视角介绍了计算机组成原理，主要以Java虚拟机为例，因为Java虚拟机是一个极为便利、时新、可移植以及几乎到处可得到的平台。

本书主张读者在Java虚拟机的范围内彻底理解计算机组成的核心原理，然后将这些原理拓展到其他四个最主要的平台：Intel 8088、Pentium 4、Power体系结构及Atmel AVR微控制器。使读者能快速掌握实际环境中计算机体系结构原理，提高实践和应用能力。

本书主要内容

- 计算、表示以及虚拟机的角色。
- 算术表达式：符号表示、存储程序计算机及运算。
- 采用领先的开源Java汇编器jasmin进行汇编语言编程。
- 从if语句和循环到子例程的控制结构。
- 真实的计算机体系结构：优化CPU、存储器及外设。
- 8088、Pentium及Power：比较其组成、体系结构及汇编语言。
- Pentium和Power体系结构的性能问题，包括流水线。
- 微控制器：组成、体系结构、接口及程序设计。
- 高级Java虚拟机编程：复杂和派生类型、类、继承、类操作、I/O等。
- 附录涵盖了数字逻辑、Java虚拟机指令集、操作代码及类文件格式。

作者简介

Patrick Juola 科罗拉多大学计算机科学博士，现为迪尤肯大学数学与计算机科学系副教授。他的研究兴趣包括自然语言处理、语言心理学及计算机安全。他曾在牛津大学做博士后，在卡内基－梅隆大学的CERT/CC做访问科学家，在PGP公司做专职科学家。

HALL www.PearsonEd.com

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com



上架指导：计算机 计算机组成

ISBN 978-7-111-27785-9

9 787111 277859

定价：39.00元

计 算 机 科 学 丛 书

计算机组成及汇编语言原理

(美) Patrick Juola 迪尤肯大学 著 吴为民 艾丽华 张大伟 译

Principles of Computer Organization and Assembly Language

Using the Java Virtual Machine

机械工业出版社
China Machine Press

本书以Java虚拟机为基础介绍计算机组织和系统结构。前半部分涵盖了计算机组织的一般原理，以及汇编语言编程的艺术，后半部分关注于各种不同CPU在系统结构上的特殊细节，包括奔腾、8088、Power系统结构以及作为典型嵌入式系统控制芯片例子的Atmel AVR。

本书全面反映了IEEE和ACM所推荐的标准计算机体系结构及组成课程应涵盖的知识要点，适用范围广，可作为高等院校计算机及相关专业计算机组成课程的教材。

Simplified Chinese edition copyright © 2010 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Principles of Computer Organization and Assembly Language: Using the Java Virtual Machine* (ISBN 0-13-148683-7) by Patrick Juola, Copyright © 2007.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall.

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究
本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2007-1831

图书在版编目（CIP）数据

计算机组成及汇编语言原理 /（美）卓拉（Juola, P.）著；吴为民等译. —北京：机械工业出版社，2009.9

（计算机科学丛书）

书名原文：Principles of Computer Organization and Assembly Language: Using the Java Virtual Machine

ISBN 978-7-111-27785-9

Ⅰ.计… Ⅱ.① 卓… ② 吴… Ⅲ.① 计算机体系结构 ② 汇编语言 Ⅳ.TP303 TP313

中国版本图书馆CIP数据核字（2009）第123710号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）
责任编辑：刘立卿
北京市荣盛彩色印刷有限公司印刷
2010年1月第1版第1次印刷
184mm×260mm · 15.75印张
标准书号：ISBN 978-7-111-27785-9
定价：39.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：(010) 88378991，88361066
购书热线：(010) 68326294，88379649，68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章分社较早意识到"出版要为教育服务"。自1998年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以"计算机科学丛书"为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

"计算机科学丛书"的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，"计算机科学丛书"已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版"经典原版书库"作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章分社欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



译者序

当前，对于计算机组成与系统结构类的本科课程，在教学上的主要困难之一是难以选择一个合适的教学用体系结构。能清楚体现计算机组成和体系结构原理的芯片早已过时；而对于先进的奔腾机，这些基本原理则淹没于复杂的实现方法和策略中。

本书作者意识到了目前计算机组织和系统结构在教学选材上的困难，并采取JVM作为教学体系结构。这是从新的角度进行的有益尝试。JVM非常简单、易于理解，因而可能会成为系统结构教学的最佳用机之一。但JVM毕竟与真实计算机存在物理差别，为表明这种差别，作者也有针对性地介绍了其他几种典型的体系结构。

本书的特点是内容广泛且有一定深度，从最基本的电子器件、二进制表示和计算，到jasmin汇编语言程序设计，再到现实世界中存在的计算机系统结构，最后到JVM高级编程课题，几乎涵盖了所有相关的主题。并且，在每个章节都提供了习题，以巩固知识。

本书适合于作为大学二、三年级相关课程的教材或教学参考书。学生们通过一学期的学习，就能基本掌握计算机组成的基本原理及汇编语言编程。当然，如果学生们已经掌握了计算机的最基础知识，再学习本书则效果更好。

本书由三位老师合作翻译。吴为民翻译了第1、2、3、4、10章以及附录A、C、D、E，艾丽华翻译了第5、6、7、8、9章，张大伟翻译了附录B。由于本书的翻译工作是在繁忙的教学、科研工作之余完成的，难免有疏漏之处，欢迎各位读者给予批评指正。

译者

2009年10月



前　言

本书内容

这是一本关于Java虚拟机（Java Virtual Machine，JVM）组织和系统结构的书。JVM是处于Java语言核心的软件，并出现在大多数计算机、Web浏览器、PDA以及网络化附属设备中。本书还涵盖了计算机组织和系统结构的一般原理，并以其他流行（或不那么流行）的计算机为例加以说明。

这不是一本关于编程语言Java的书，虽然具备Java语言或类Java语言（C、C++、Pascal、Algol等）的一些知识会有所帮助。本书是一本关于Java语言如何使事件发生以及计算如何产生的书。

这本书的写作开始于一个现代技术的实验。当我开始任教于目前的大学时（1998年），计算机组织和系统结构课程用的主要是运行MS-DOS的8088，这个编程环境实质上与修这门课的二年级学生年龄相当。（遗憾的是，这种时间上的迟滞相当普遍。当我在本科修同样的课程时，所学系统结构的相应计算机只比我"年轻"2年。）根本问题是现代奔腾4芯片不是特别好的教学用系统结构。它加入了有20年历史的8088的所有功能，包括其局限，并提供了复杂的变通方法。由于这个复杂性问题，就难以在不详细引用早已过时的芯片集的情况下解释清楚奔腾4的工作原理。教科书主要讲解的是较简单的8088，然后作为扩展和后续思考来描述实际要使用的计算机。这就好比在福特A型上学习汽车力学，后来只讨论如催化式排气净化器、自动驾驶、基于钥匙的点火系统等重要概念。计算机系统结构课程不应被迫成为计算历史的课程。

与此不同的是，我想采用一种易于理解的系统结构来教这门课，该系统结构结合了现代原理且本身对学生有用。由于每个运行Web浏览器的计算机都结合了JVM的一个副本作为软件，因此几乎每个当今的计算机都已经有了兼容的JVM供其使用。

因而这本书涵盖了计算机组织和系统结构的核心方面：数字逻辑和系统、数据表示以及计算机组织/系统结构。本书还描述了一种特定系统结构JVM的汇编级语言，并且介绍了其他常见的系统结构（如英特尔奔腾4和Power PC）作为支持例子但不作为重点。正如IEEE计算机学会和美国计算机协会所推荐的，本书尤其适合作为计算机系统结构和组织的标准二年级课程。⊖

组织

本书包含两个部分。前半部分（第1～5章）涵盖了计算机组织和系统结构的一般原理，以及汇编语言编程的艺术/科学，并采用了JVM作为例子来阐明这些原理如何起作用（在数字计算机中如何表示数？加载器做哪些事情？格式转换涉及哪些事情？），以及JVM汇编语言编程中一些必要的细节，包括对操作代码的详细讨论（操作代码i2c要做哪些事情，它是如何改变堆栈的？运行汇编器的命令是什么）。本书的后半部分（第6～10章）关注于各种不同CPU在系统结构上的特殊细节，包括奔腾、它的老亲戚8088、Power系统结构，以及作为典型嵌入式系统控制芯片例子的Atmel AVR。

⊖ "Computing Curricula 2001," 2001年12月15日，最终草案，特别参见关于课程CS220的推荐意见。

读者

这个框架将使得本书被广大读者和众多课程所使用，这是我的希望和信念。本书应能成功地服务于以软件为中心的计算机产业。对于那些主要感兴趣于将编程语言作为基础来学习抽象的计算机科学的人来说，JVM对计算的基本操作提供了一个简单、易于理解的介绍。作为编译器理论、编程语言或操作系统课程的基础，JVM是一个便利和可移植的平台和目标系统结构，比任何单芯片或操作系统有更广的可用性。作为进一步学习（特定平台的）各种计算机的基础，JVM提供了一个有用的解释性教学系统结构，该系统结构不仅可向目前的奔腾，而且可向在未来可能取代或支持奔腾的其他系统结构，实现平滑的、有原则的过渡。对于有兴趣学习计算机如何工作的学生来说，本书将提供有关大量不同平台的信息，以增强使用实际计算机和系统结构的能力。

如上所述，本书主要是作为本科二年级的单学期课程的教科书。前四章给出了理解计算机组织、系统结构以及汇编语言编程所需的核心材料。假设读者已经有了高级命令性语言的一些知识，并且熟悉高中代数（不是微积分）。在此基础上，教授（和学生）在选择主题方面有某种程度的灵活性，这取决于环境和具体问题。对于Intel/Windows工作组，关于8088和奔腾的章节就是有用和相关的，而对于有老式苹果机或基于Motorola微处理器实验室的学校，关于Power系统结构的章节更为相关。讲述Atmel AVR的一章可为嵌入式系统或微计算机实验室工作奠定基础，而高级的JVM课题将是打算以JVM系统结构为基础实现基于JVM的系统或编写系统软件（编译器、解释器等等）的学生之兴趣所在。进度快的课程甚至可能会涵盖本书所有的主题。书中还提供了附录供参考，因为我们相信，好的教科书应该在课程结束后仍是有用的。

致谢

没有Duquesne大学的学生，尤其是在计算机组织和汇编语言课程中参加我的实验的学生们，就不会有这本书。还要感谢我所在的系、学院以及大学所提供的帮助，尤其是来自Philip H.和Betty L. Wimmer家庭基金会的基金支持。我还要感谢我的读者，尤其是Pittsburgh大学的Erik Lindsley对早期草稿的宝贵意见。

没有出版商，本书永远不会与读者见面。因此我还要感谢Tracey Dunkelberger和Kate Hargett两位编辑，并通过他们向Prentice Hall出版集团致谢。我要向所有的审阅人致以谢意：Western Illinois大学的Mike Litman、Texas Tech大学的Noe Lopez Benitez、Arkansas Tech大学的Larry Morell、加州州立大学（Channel Islands）的Peter Smith、路易斯安娜州立大学（Shreveport）的John Sigle、以及密苏里大学（Columbia）的Harry Tyrer。同样，没有软件也不会有这本书。除了显然要感谢Sun公司发明Java的那些人以外，我特别地想要感谢jasmin的作者Jon Meyer，感谢他编写的软件以及他提供的有益支持。

最后，我还要感谢我的妻子Jodi，她为大多数示意图绘制了最初草图。更重要的是，在本书的长期写作过程中，她一直在努力容忍我，并且仍然愿意与我生活在一起。

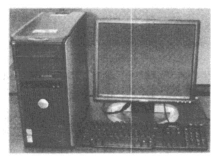
目录

第二部分 真实计算机

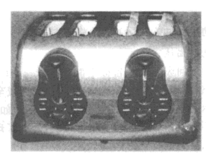
第一部分　假想计算机

第1章　计算和表示

1.1　计算



一台计算机



也是一台计算机

1.1.1　电子设备

有多少人真正知道计算机是什么？如果你问这个问题，大多数人会指向某人桌子上（或者也许是某人公文包中）的一组盒子——这可能是一组由灰色塑料包装的、外形呆板的方形盒子，并且纠缠了一堆连线，类似于一台电视机。如果穷追细节，他们会指向某个盒子，称之为"计算机"。不过当然也有计算机是隐藏在各种日常电子部件内部的，它们的作用可能是确保汽车的燃油效率足够高，解释来自DVD播放机的信号，甚至是确保早餐面包烤得恰到好处。但是对于大多数人来说，计算机仍然是你从电子商店购买的盒子，并且还要常常比较其存储量（如位数和字节数）和频率（如千兆赫），但很少有人真正明白其含义。

用功能的术语来说，计算机就是一台高速的计算器，平均每秒能执行几千、几百万，甚至几十亿的简单算术操作，这些操作由存储的程序所规定。大概每千分之一秒左右，汽车中的计算机就会从发动机中的各个传感器读取一些关键的性能指示数据，并对汽车进行微调以确保运转正常。该功能的关键至少有某些部分是在传感器中，计算机本身只处理电信号。传感器负责确定发动机究竟运转状况如何，并将这些信息转换成一组电信号，用以描述或表示发动机的当前状态。类似，计算机所做的调节被存储为电信号，并被转换成为发动机工作状况的实际变化。

电信号如何能"表示"信息？计算机如何精确地处理这些信号，以达到精细的控制而无需任何人的干涉？这种表示问题就是理解计算机如何工作以及如何在现实世界中部署计算机的关键。

1.1.2　算法机

计算机操作方面的最重要的概念就是算法（algorithm）：算法就是一个明确的、按步进行

的过程，用以解决某个问题或达到某个期望目标。计算机的最终定义不依赖于其物理特性，甚至也不依赖于其电学特性(如其晶体管)，而是依赖于其表示和完成算法的能力，这些算法来源于存储的程序。在计算机内部是数以百万计的微小电路，每个电路在被调用时都执行一个特定的、明确定义的任务（如将两个整数相加、使单个或一组线通电）。大多数使用计算机或者为计算机编程的人都不知道这些电路的工作细节。

特别是，一个典型的计算机能执行若干个基本类型的操作。由于计算机从根本上看只是计算机器，所以它能执行的几乎所有功能都与数字（以及用数字表示的概念）相关。一个计算机通常能执行诸如加法和除法等基本的数学操作。它也能执行基本的比较操作，如一个数字与另一个数字相等吗？第一个数字小于第二个数字吗？它能存储几百万或几十亿的信息片断，并能单独地获取。最后，它能根据获取到的信息和执行比较的结果来调整其动作。比如，如果获取到的值大于以前的值，则说明发动机正在过热的情况下运行，需要发一个信号来调节其性能。

1.1.3 功能部件

系统级描述

几乎任何大学的公告板上都有这样一些广告，如“超级计算机！ 3.0-GHz Intel Celeron D, 512mg, 80-GB硬驱，15英寸显示器，为抵汽车款忍痛割爱！”。像大多数广告一样，许多信息需要深入地解读才能理解其全部意思。例如，15英寸显示器的哪个部分才真正是15英寸？(显示屏对角线的长度，够奇怪的了)。为了理解计算机的工作细节，我们首先必须理解主要的部件以及它们相互之间的关系（见图1-1)。

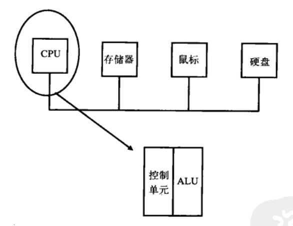


图1-1 一台计算机的主要硬件部件

中央处理单元

计算机的心脏就是中央处理单元（Central Processing Unit)，即CPU。这通常是制造在单个集成电路（Integrated Circuit, IC）硅片上的一个高密度电路（见图1-2)。它通常看起来就像一小块硅片，安置在一个几平方厘米并由金属管脚包围着的塑料厚片上。塑料厚片本身座落在一个母板(motherboard)上，母板就是一个电子电路板，由一块塑料和金属组成，每个面几十平方厘米，包含了CPU和其他一些部件，这些部件因速度和便利等原因而需要安置在靠近CPU的地方。CPU是计算机的最终控制器，也是执行所有计算的地方。当然，这也是人们

谈论计算机时所指的部分，比如："3.60GHz奔腾4计算机，如惠普HP xw4200"，就是指这样的一台计算机：它的CPU是奔腾4芯片，运行速度为3.6千兆赫兹(GHz)，即每秒3 600 000 000个机器周期(machine cycle)。计算机的多数基本操作需要一个机器周期，所以另外一种描述方式就是3.60GHz计算机能在每秒执行超过35亿个基本操作。在写本书的时候，3.6GHz是速度很快的机器，但随着工艺进步，情况变化很快。例如，在2000年，1.0GHz奔腾是最先进的，根据计算能力每18个月翻一番这一长期证明有效的经验（摩尔定律），我们可以预测8GHz CPU在2008年将会普及。

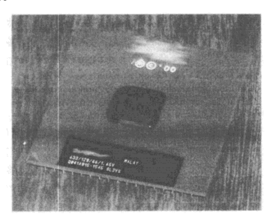


图1-2 CPU芯片的照片

CPU通常以工艺发展的系列来描述：例如，奔腾4是奔腾、奔腾2及奔腾3的进一步发展，都是Intel公司生产的。在这之前，奔腾本身衍生于一长串用数字编号的Intel芯片，开始于Intel 8088，并发展到80286、80386及80486。这个所谓的"x86系列"就成为最畅销的IBM个人计算机（PC机及其模仿机）的基础，并且可能是最广泛使用的CPU芯片。现代苹果计算机采用了一个不同系列的芯片，即PowerPCG3和G4，该系列芯片由苹果、IBM及Motorola组成的联盟（AIM）所制造。较早的苹果和Sun工作站采用了Motorola设计的68000系列芯片。

CPU本身可分为两个或三个主要的功能部件。控制单元(Control Unit)负责在计算机内移动数据。例如，控制单元要从存储器中装入单个程序指令，辨别各指令的功能，并将指令传递到计算机的其他部分来执行。算术和逻辑单元（ALU）为计算机执行所有必需的算术运算。它通常包含完成加法、乘法、除法等的特殊用途硬件。顾名思义，它还执行所有的逻辑操作，确定一个给定的数是否大于或小于另外一个数，或者检查两个数是否相等。一些计算机（特别是较早的）有专用的硬件，有时安置在与CPU不同的芯片上，用于处理涉及分数和小数的操作。这个特殊的硬件通常称为浮点单元或FPU（也称为浮点处理器或FPP）。其他计算机将FPU硬件放在ALU和控制单元所在的同一个芯片上，但FPU仍可看作是同一电路系统内部的不同模块。

存储器

要执行的程序和其数据都存放在存储器中。在概念上，存储器可看作是一个非常长的一列或一排的电磁存储器件。这些列的位置从0到CPU定义的最大数进行编号，可由控制单元单独地寻址，将数据置入存储器或将数据从存储器中取回（图1-3）。此外，多数的现代计算机都允许像磁盘驱动器这样的高速设备拷贝大块的数据而无需对每个信号都要控制单元介入。

存储器可宽泛地分为两种类型：只读存储器（ROM），这是永久的、不可改变的，甚至在电源关闭后数据仍保留；随机存取存储器（RAM），其内容可被CPU改变，是作为临时存储器但通常在电源关闭后数据消失。很多机器中同时有这两种存储器。ROM保存标准化的数据和操作系统基本版本，用于启动机器。更多程序保存在像磁盘驱动器和CD这样的长期存储器中，并在需要时被装入RAM中用于短期存储和执行。

补充资料

摩尔定律

Intel的共同创始人戈登·摩尔在1965年观察到，能放置到一个芯片上的晶体管数量每年翻一番。在20世纪70年代，这个步伐稍微变慢了，成为每18个月翻一番。但令每个人包括摩尔博士本人吃惊的是，从此这个步伐就异乎寻常地均匀。仅仅在刚过去的几年这个步伐才减慢。

更小的晶体管（相应的晶体管密度更大）所蕴涵的意义是深远的。首先，硅片本身每平方英寸的成本比较而言已经相对稳定了，所以密度加倍就使芯片的成本近似减半。其次，更小的晶体管反应更快，并且部件能更紧密地放置在一起，所以它们能更快地相互通信，极大地提高了芯片速度。较小的晶体管也消耗较少的功耗，这意味着更长的电池寿命和较低的散热要求，避免了对房间气温控制及庞大电扇的需求。由于更多的晶体管可放置到一个芯片上，就需要较少的焊接以将芯片连接到一起，因而就降低了焊接破损的机会，相应地也就极大地提高了总体可靠性。最后，芯片更小的事实意味着计算机本身可以做得更小，小到足以使嵌入式控制芯片和/或个人数字助理（PDA）这些思想成为现实。很难预测摩尔定律对现代计算机的发展产生多大影响。到目前，摩尔定律通常就是简单地用来表明计算机的能力在每18个月翻一番（不管什么原因，不仅仅是晶体管密度）。一个当地商店下架的标准的、甚至低端的计算机就比1973年的原始Cary-1超级计算机更快、更可靠，并且有更多的内存。

摩尔定律的问题是它不会永远保持。最终，物理定律指出，晶体管不能小于一个原子（或类似的东西）。更令人不安的是摩尔第二定律指出制造成本在每3年翻一番。只要制造成本增加的速度比计算机能力增长的速度慢，性能/成本比就应该保持合理。但是，投资新的芯片工艺的成本很大，可能会使Intel这样的制造商继续投入新的资本变得困难。

这个简化的描述故意隐藏了存储器的一些繁杂方面，通常硬件和操作系统为用户处理这些方面。（这些问题也趋向于与硬件相关，所以将在后面章节详细讨论。）例如，不同的计算机，即使有相同的CPU，也常常有不同数量的存储器。安置在计算机上的物理存储器可少于CPU能寻址的最多位置数，但在特定情况下，却可能多于CPU能寻址的最多位置数。进一步地，处于CPU芯片本身的存储器通常可以比处于不同芯片上的存储器以快得多的速度访问，所以，一个好的系统应努力确保数据需要移动或拷贝时能在最快的存储器中得到。

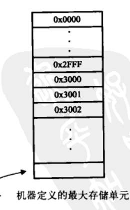


图1-3 线性排列的存储器单元的示意图

输入/输出（I/O）外设

除了CPU和存储器，一台计算机通常还包含其他

设备用来读、显示或存储数据，或者更一般地与外部世界交互作用。这些设备多种多样，从普通的键盘和硬盘驱动器，到并不普通的设备如传真（FAX）板、话筒及音乐键盘，再到相当怪异的小配件像化学传感器、机器人手臂及安全门拴。这些部件的一般术语就是外设(peripheral)。在很大程度上，这些设备对计算机本身的体系结构和组织几乎没有直接的影响，它们只是信息的源点和终点。例如，键盘只是一个让计算机从用户那里获取信息的设备。从CPU设计者的角度看，数据就是数据，不管它来自于因特网，来自于键盘，还是来自于奇特的化学频谱分析仪。

在很多情况下，一个外设可在物理上分为两个或更多的部分。例如，计算机通常在某种形式的视频监视器上将其信息显示给用户。监视器（monitor）本身就是一个独立的设备，通过电缆连接到计算机机箱内部的视频适配板上。CPU画图时，可发送命令信号给视频板，视频板生成图形并通过视频电缆将适当的视觉信号发送到监视器本身。可以用类似的过程描述计算机如何通过一个SCSI（Small Computer System Interface）控制器卡从很多不同种类的硬盘驱动器装入一个文件，或者通过一个以太网卡与构成因特网的数百万英里的线交互作用。在概念上，工程师们会对设备本身、设备电缆（通常只是一条线）及设备控制器（通常是计算机内部的一块板）加以区分，但对于程序员，它们通常从总体上就被看作是一个设备。根据这种逻辑，整个因特网连同其数百万英里的线就只是“一个设备”。对于一个设计良好的系统，从因特网下载一个文件与从一个硬盘驱动器装入一个文件之间没有多大区别。

互连和总线

为了使数据在CPU、存储器以及外设之间移动，必须存在连接。这些连接（特别是独立的板之间的连接）通常是成组的线，使得多个单独的信号能成组发送。例如，最初的IBM-PC有8条线在CPU和外设之间传递数据。一台更现代的计算机的PCI（Peripheral Component Interconnect）总线有64条数据线，即使不考虑计算机速度的提高，也能使数据以8倍的速率传递。这些线通常组合起来形成总线（bus），总线就是连接若干不同设备的线的集合。由于总线是共享的（就像早期的共线电话），一次只有一个设备能传送数据，但连接的所有设备都可得到数据。一些附加信号用于确定哪个设备应该得到数据以及当它得到数据时应该做些什么。

一般来说，连接到单个总线的设备越多，运行就越慢。这有两个原因。首先，设备越多，在相同时间两个设备要同时传送数据的可能性就越大，因此一个设备就要等待轮到自己传送。其次，更多的设备通常意味着总线更长，由于传播有延迟（即信号从线的一端到达另一端的时间），因而就降低了总线的速度。由于这个原因，很多计算机现在采取多总线设计，例如，局部总线连接CPU和CPU母板上的高速存储器（通常置于CPU芯片本身），系统总线连接存储器板、CPU母板以及一个“扩展总线”接口板，而扩展总线又是一个连接网络、磁盘驱动器、键盘及鼠标的二级总线。

在个别的高性能计算机上（如图1-4所示的计算机），可能有4到5条独立的总线，一条保留用于高速、数据密集的设备，如网络和视频卡，而低速设备如键盘则归入另外的低速总线处理。

支持单元

除了已经提到的设备，一个典型的计算机还有一些对保证其物理条件很重要的部件。例如，在机箱内部（机箱本身也是对易损电路板的重要物理保护）有一个供电电源，用于将交流电压转换成可用于电路板的适当调节的直流电压。还可能有一个电池，尤其在便携式电脑中，用于在没有墙壁插座提供电流时提供电源并保持存储器的设置。通常还有一个风扇用来

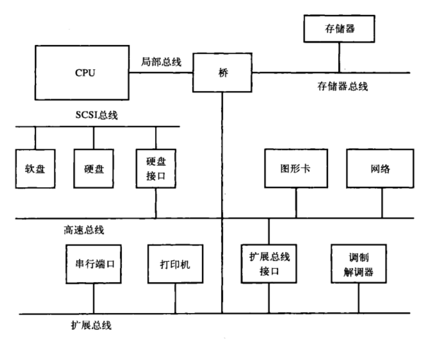


图1-4　高速多总线计算机的体系结构示意图

对机箱内部通风以防止部件过热。可能还有其他一些设备，如热传感器（用于控制风扇速度）、用于防止未授权的使用和拆卸的安全设备，常常还有若干个完全内部使用的外设如内部磁盘驱动器和CD阅读器。

1.2　数字和数值表示

1.2.1　数字表示和位

在最基本的层次，计算机部件像很多其他电子部件一样有两个状态。灯或者开或者关，开关或者打开或者关闭，线路或者正携带电流或者未携带电流。对于计算机硬件，各个部件如晶体管和电阻器相对于地或者处于0电压或者处于某个其他电压（典型值是对地有5伏特电压）。这两个状态通常用来分别表示数字1和0。在计算机发展的早期阶段，这些值是通过翻转机械开关来进行手工编码的。现在，高速晶体管实现的是几乎同样的意图，但数据用这两个值来表示自上世纪四十年代以来一直未改变。每个这样的1或0通常称为位或比特（bit），是“binary digit”的简写。（当然，“bit”本身是一个标准的英文词，意思是“一个很小的量”，这也描述了“一小部分”的信息）。

补充资料

晶体管如何工作

在现代计算机中最重要的电子部件是晶体管，晶体管是由Bardeen、Brattain和Shockley于1947在贝尔电话实验室发明的。（这些人因为这项发明而获得了1956年的诺贝尔物理学奖）。它的基本思想涉及一些相当高级的（是的，诺贝尔档次的）量子物理学，但是，不需要具体的公式，你也能根据电子迁移的原理来理解它。一个晶体管主要包含了

一种称为半导体（semiconductor）的材料，它处于好导体（如铜）和坏导体/好绝缘体（如玻璃）之间的不稳定的中间状态。半导体的一个关键方面是其导电能力会随半导体中杂质（掺杂物，dopant）的不同而显著变化。

例如，当元素磷被加入到纯硅（一种半导体）时，将为硅提供电子。由于电子带负电，磷就称为n型（n-type）掺杂物，而用磷掺杂过的硅就叫做n型半导体（n-type semiconductor）。与此相反，铝是一种p型（p-type）掺杂物，从硅阵列中移出（实际上是锁定）电子。在p型半导体中，这些电子被移出的地方有时称为“空穴”。

当将一块n型半导体与一块p型半导体紧挨着放在一起时（所形成的小器件称为二极管(diode)，见图1-5），就会发生有趣的电效应。电流一般不能穿过这样的二极管，载流的电子将遭遇并“陷入”空穴。然而，如果你将一个偏置电压（bias voltage）施加于这个器件，额外的电子将填充空穴，使得电流通过。这意味着电流只能在一个方向穿过二极管，使得其具有电整流器（rectifier）的用途。

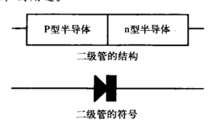


图1-5 二级管

现代的晶体管做得就像一个半导体三明治，就是一个p型半导体薄层处于两薄片的n型半导体之间。(是的，这就是两个背靠背的二极管，见图1-6)。在常态环境下，电流不能从发射极(emitter)到集电极（collector）穿过，因为电子陷入到空穴中。将偏置电压施加到基极（base)（中间的线）就将填充空穴使得电流能通过。你可以将基极看作一个可开关的门，使得电流能流过或不能通过。或者，你也可以将其看作是橡胶软管中的一个阀门，用以控制通过的水流量。将阀门转到一个方向，电信号就减小到很微弱，转到另一个方向，就无阻碍地流动。

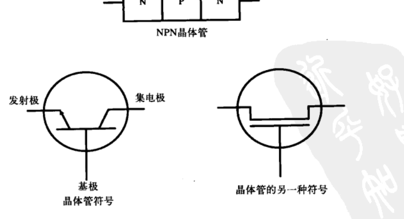
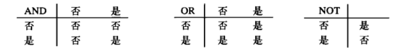


图1-6 晶体管

晶体管的总体效应是（在基极）电压的小的变化将导致从发射极到集电极流过电流量的很大变化。这使得晶体管在放大小信号方面极为有用。晶体管还可作为二进制开关使用，其关键优势是没有移动部件因而不会有损坏。利用集成电路的发明，Jack Kilby也因此赢得了诺贝尔奖，工程师们通过在更大硅片上利用很小块的区域而获得了生成数千、数百万、或数十亿微小晶体管的能力。直至现在，这仍然是制造计算机的主要方式。

1.2.2 布尔逻辑

现代计算机采用基于位（bit）的逻辑来操作。一个位是就是携带信息的最小单位，就像儿童游戏“二十个问题”中的问题，每个问题的答案为“是”或者“否”，这样每个问题的答案都可编码成一个位（例如，1表示“是”，0表示“否”）。如果你熟悉早期的Milton Bradley Board游戏“猜猜我是谁”，也是同样一个道理。如果询问是/否的问题，你就逐渐知道了你的对手选择了谁，并且通过记录问题的答案，你可在任何时候再现学习到的东西。因此，一个位就是可处理（存储、传送或操作）的最小单位。对用位表示的量进行操作的传统方式称为布尔（Boolean）逻辑，这是以十九世纪数学家乔治·布尔来命名的。他确定了基本的操作：AND（与）、OR（或）和NOT（非），并以对位的简单变化定义了它们的含义。例如，表达式X AND Y为真（也可称为“是”或者1）当且仅当X和Y都为“是”。相反地，倘若表达式X OR Y为“是”，只要X为“是”或者Y为“是”即可。与此等价的陈述是：X OR Y为假（“否”或者0）当且仅当X和Y都为“否”。表达式NOT X就是X的反面：当X为“否”时它为“是”，X为“是”时它为“否”（见图1-7）。因为一个位只能处于两种状态之一，所以无需考虑其他的可能性。这些操作（AND、OR及NOT）可根据需要嵌套或结合。例如，NOT（NOT（X））就是对X取反后再取反，结果就是X本身。这三种操作与它们对应的英文词汇有很好的对应关系：如果我想要一杯“有牛奶和糖”的咖啡，在逻辑上，我想要的就是一杯“有牛奶”条件为真并且“有糖”条件为真的咖啡。类似地，一杯“没有牛奶或糖”的咖啡与一杯“没有牛奶并且没有糖”的咖啡是一个意思。（可以对这个问题多做一些考虑。）



AND	否	是
否	否	否
是	否	是

OR	否	是
否	否	是
是	是	是

NOT	
否	是
是	否

图1-7　AND、OR及NOT的真值表

除了这三种基本的操作，还有根据这些基本操作定义的一些其他操作。例如，NAND是NOT-AND的缩写。表达式X NAND Y就是指NOT（X AND Y）。类似地，X NOR Y就是指NOT（X OR Y）。另一个常见的表达式是异或（exclusive-OR），写成XOR。表达式 X OR Y为真，则X为真，或Y为真，或者两者都为真。对比之下，X XOR Y为真，则X为真，或Y为真，但不能两者都为真。这个差异在英文中没有明确地表达出来，而隐含在一些不同的用法中。例如，如果我被问到我的咖啡中是否需要牛奶或糖，我可能说“好的”，意思是我两者都要。这是正常的（inclusive）OR的意思。与此不同的是，如果问我要咖啡还是茶，我若说“好的”来表示两者都要就不很恰当。这是一个(exclusive) OR，我能要咖啡XOR茶，但不能同时两者都要。

从严格的理论观点看，将1/“是”/“真”编码为地电压还是编码为对地5伏电压没有多大关系，只要将0/“否”/“假”总是编码成与之不同的值即可。从计算机工程师或系统设计者的观点看，就可能有特定的原因（如功耗）来选择一种表示而不选择另一种。表示法的选择对芯片设计本身会有深远的影响。上面描述的布尔操作通常在芯片非常低的层次上用硬件实现。例如，

我们可以用一对开关（晶体管）建立一个简单的电路，使得只有在两个开关都关闭的情况下电流才能流过。这样一个电路称为与（AND）门，因为它对表示成开关状态的2个位实现了AND的功能。这个微小电路与其他类似的电路（OR门，NAND门等等）一样，在计算机芯片上要复制几百万或几十亿次，是计算机的基本构件块。（关于这些门如何工作的更多信息参见附录A。）

1.2.3 字节和字

为方便起见，8个位通常组合成一块，传统上称为字节（byte）。这样做主要有两个优势。首先，对于人来说，读写一个长长的0/1序列很乏味且容易出错。其次，大多数有意义的计算都要求多位的数据。如果有像标准总线那样的多条线，则电信号可成组移动，实现更快的计算。

下一个命名位块称为字（word）。字的定义和大小不是绝对的，而是随计算机的不同而变化。一个字就是计算机最便于处理的数据块大小。（虽然不总是如此，但通常都是通向主存储器的总线大小。但是后面要讨论的Intel 8088是一个反例。）例如，Zilog Z-80微处理器（作为Radio Shack TRS-80的基础芯片，在20世纪七十年代中期流行）的字大小是8位，即一个字节。CPU、存储器及总线都已优化为一次处理8位。（例如，系统总线中有8条数据线）。当计算机必须要处理16位数据时，就要分两部分分别处理。而如果计算机只有4位数据要处理时，CPU就像处理8位数据一样工作，然后抛弃额外4个无用的位。原始的IBM PC是基于Intel 8088芯片的，有16位大小的字。更现代的计算机如Intel 奔腾4或者PowerPC G4有32位大小的字。有64位或更大字的计算机如Intel Itanium和AMD Operon系列也已经出现了。尤其对于高端科学计算或图形学，如家庭视频游戏控制台，字的大小会成为能否以足够快的速度传送数据以支持平滑变化的高精细度图形的关键。

正式地讲，计算机字的大小定义为其寄存器（register）的大小（用位数表示）。寄存器是CPU内部实际进行如加、减、及比较等计算的存储位置（图1-8）。寄存器的数量、类型及组织对于不同的芯片变化很大，甚至对同一系列的芯片变化也会很大。例如，Intel 8088有4个16位通用寄存器，而7年后设计的Intel 80386则采用了32位寄存器。对寄存器高效的利用是编写快速、优化的程序的关键。遗憾的是，由于不同计算机之间存在差异，使得这成为比较困难的一个方面。

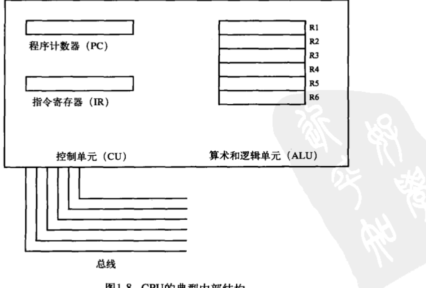


图1-8 CPU的典型内部结构

1.2.4 表示

位模式的解释是任意的

首先考虑一下老式的8位微计算机芯片中的寄存器。它能保存多少种不同的模式呢？对这个问题的另一种问法是：按正反面排列8个便士硬币的序列有多少种方式，或者，分别是0或1的8个数字能构成多少种串。

对于第一个位/硬币/数字有两种可能性，对第二个也有两种可能性，这样一直下去直到我们达到第8位即最后一位数字。这样就有$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$种可能性，算出来就是$2^8=64$种不同的存储模式。类似的推算显示出，对于32位寄存器，存在2^{32}即刚好超过40亿种存储模式。(的确，对于过分苛刻的人来说应该是$2^{32}=4\ 294\ 967\ 296$。处理大的二进制幂的一个有用经验是2^{10}虽然真实值是1024，但十分接近于1000。回忆一下，数的相乘可通过指数相加实现：$a^b \cdot a^c=a^{b+c}$。这样，2^{32}就是$2^{2+10+10+10}$，即$2^2 \cdot 2^{10} \cdot 2 \cdot {}^{10} \cdot 2 \cdot {}^{10}$，也就是大约$4 \cdot 1000 \cdot 1000 \cdot 1000$。)

但是这些模式意味着什么呢？实际的答案是：要由程序员来决定它们意味什么。在下一个小节你将会看到，可以将位模式00101101解释为数字77。也有可能解释成8个不同的是/否问题的答案。(“你结婚了吗？”－没有。“你过了25岁了吗？”－没有。“你是男性吗？”－是。如此等等。）它也可以表示键盘上所按下的键（在这种情况下，这个模式就表示大写的M的ASCII值）。对位模式的解释是任意的，对同一种模式，计算机可以有很多种使用方式。程序员的一部分任务就是确保计算机总是能正确地解释这些任意的、意义含糊的模式。

自然数

解释位模式的常用方式涉及二进制（binary）算术运算（基数为2）。在传统的十进制（decimal）运算（基数为10）中，只有从0到9这十个不同的数字符号。更大的数就表示为单独的数位乘以某个基数值的幂。例如，数字481就表示为$4 \cdot 10^2+8 \cdot 10^1+1$。采用这种表示法，我们只用三位十进制数字就能表示999以内的所有自然数。采用类似的表示法，但用2的幂累加，我们就能用0和1表示任何二进制数。

以十进制数85作为例子，通过简单的运算显示出该数等于$64+16+4+1$，更详细地写成$1 \cdot 2^6+0 \cdot 2^5+1 \cdot 2^4+0 \cdot 2^3+1 \cdot 2^2+0 \cdot 2^1+1$。采用二进制，这个数就写成1010101。用8位寄存器，这个数可存储为01010101；而用32位寄存器，将成为000000000 000000000 0000000 1010 101。

可以在这些二进制数上进行算术运算，所采用的策略和算法与小学生解决10进制问题一样。事实上，二进制运算甚至更容易，因为加法和乘法表更小而且简单得多！因为只有0和1两个数字，所以表中只有四项，如图1-9所示。只要记住，当进行二进制加法（基数为2）时，只要得到结果2就产生一个进位（就像在十进制中每次得到结果10就产生一个进位一样)。这样，在基数为2时，1＋1的结果不是2，而是0并有进位1，也就是10。

+	0	1
0	0	1
1	1	10

·	0	1
0	0	0
1	0	1

图1-9　二进制（基数为2）运算的加法和乘法表

对这些表进行观察，可以发现二进制运算和布尔代数之间的本质联系。乘法表与AND是一样的。当然加法可能产生两个数：和与（可能的）进位。进位存在的充分必要条件是第一个数字和第二个数字均是1。换句话说，进位也就是两个加数的AND，而和（不包括进位）是1的条件是第一个数字是1或者第二个数字是1，但不能都是1，也就是两个加数的XOR。通过建立适当的AND和XOR门，计算机就能在寄存器的表示能力范围内对任何数字进行加法和乘法操作。

那么，一个8位寄存器能存储多大的数字呢？最小的可能值显然是00000000，代表数字0。最大的可能值就是11111111，代表255。任何在这个范围的整数都可以容易地表示成8位的量。

对于32位寄存器，最小的值仍然是0，但最大的值刚好超过42亿。

虽然计算机解释长的二进制数没有困难，但对人类来说往往有困难。例如，32位数字00010000000000000000100000000000与数字（做一下深呼吸）0001 0000 00000000000100000000000000是一样的吗？（不，它们是不同的。第一个数的两个1之间有16个0，而第二个数的两个1之间有15个0。）因为这个原因，当有必要处理二进制数字时（希望这种情况很少），多数程序员宁愿使用十六进制（hexadecimal）（基数为16）数字来代替。由于16＝2^4，每个4位的块（有时称为半字节（nybble））可表示成一个基数为16的“数字”。在16个十六进制数字中，我们熟悉其中0到9这10个数字，表示0000到1001等模式。由于我们日常的十进制数只使用10个数字，计算机科学家就增选字母A到F来代表其余的模式（1010，1011，1100，1101，1110，1111，全部的转换列表见表1-1）。目前，这几乎是人们对十六进制数的唯一使用方式：用于指定和简写（比如密码学或网络中可能用到的）长的位串值。上面两个数转换成基数为16时明显有所不同：

0001	0000	0000	0000	0000	1000	0000	0000	
1	0	0	0	0	8	0	0	= 0x10000800

0001	0000	0000	0000	0001	0000	0000	0000	
1	0	0	0	1	0	0	0	= 0x10001000

表1-1 十六进制↔二进制数字的转换

十六进制	二进制	十六进制	二进制	十六进制	二进制	十六进制	二进制
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

根据很多计算机语言（包括Java、C、及C++）的惯例，写十六进制数时用“0x”或“0X”开头。我们在这里遵循这个惯例，这样数字1001就是指十进制数1001。值0x1001就是指16^3＋1，就是十进制值4097。（本书中的那些二进制量很好辨别。另外，在很少的情况下，一些模式写成8进制（octal）数字，即基数为8。在C、C++、或Java中，这些数字用0开头，数字01001作为八进制数就等价于513。这是一个不太走运的惯例，因为几乎没人出于某种理由使用八进制，但传统如此。）注意不论基数是什么，0还是0（1也还是1）。

基数转换

从一种基数表示转换到另一种表示可能很乏味但却是必要的事情。幸运的是，所涉及的数学知识相当简单。例如，如果你理解了表示方法，从其他任何进制转换到十进制就很简单。例如，二进制数110110就表示$2^5+2^4+2^2+2^1$，就是32＋16＋4＋2，得54。类似地，0x481就是$4\cdot16^2+8\cdot16^1+1$，就是1024+128+1，即十进制数1153。

完成计算的一种更简单的方式是交替进行乘法和加法。二进制数110110显然是二进制值11011的2倍。（如果不是显然的，就注意一下十进制数5280是值528的10倍）。接下来二进制数11011又是1101的2倍再加1。这样，简单地交替乘以基数值和加1就行了。采用这种方法，二进制110110就成为：

$$((((((((\mathbf{1}\cdot2)+\mathbf{1})\cdot2)+\mathbf{0})\cdot2)+\mathbf{1})\cdot2)+\mathbf{1})\cdot2)+\mathbf{0})$$

简单计算证实得数是54。类似地，0x481是：

$$((((\mathbf{4} \cdot 16) + \mathbf{8}) \cdot 16) + \mathbf{1})$$

可求得值为1153。

如果交替进行乘法和加法就能转换为十进制数，顺理成章地，交替进行除法和减法就能将十进制数转换到二进制数。在我们进行除法操作时，减法实际上是隐含的。当一个整数除以另一个整数时，很少得到除尽的答案。一般会有一个余数，这个余数必须隐含地从被除数中减去。这些余数就是得数。再以54为例，我们重复地除以2就产生所需要的二进制数字：

54 ÷ 2 = 27余**0**
27 ÷ 2 = 13余**1**
13 ÷ 2 = 6余**1**
6 ÷ 2 = 3余**0**
3 ÷ 2 = 1余**1**
1 ÷ 2 = 0余**1**

黑体数字当然就是54相应二进制数各位上的数（二进制数110110）。仅有的棘手事情就是要记住，在乘法进行过程中，数字是按正常顺序（从左到右）进入的，所以并不奇怪，在除法过程中，数字是按反向从右到左的顺序出来的。这个过程对基数为16（或者基数为8，基数为4，或者任意基数）也适用：

1153 ÷ 16 = 72余**1**
72 ÷ 16 = 4余**8**
4 ÷ 16 = 0余**4**

最后，最常用同时也许是最重要的转换是直接（并且快速地）在二进制和十六进制之间进行转换，不管按哪个方向。幸运的是，这也是最容易的。因为16是2的4次幂，乘以16实际上就是乘以2^4。这样，每个十六进制数字就直接对应一组4位二进制数字。就像较早讨论过的，要从十六进制转换到二进制，只要简单地将每个数字替换成等价的4位二进制数字即可。要从二进制转换到十六进制（hex），就要将二进制分裂成4位的若干个组（从右边开始），并按反方向完成替换操作。全部的转换如表1-2所示。

表1-2　十六进制↔二进制数字转换（表1-1的拷贝）

十六进制	二进制	十六进制	二进制	十六进制	二进制	十六进制	二进制
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

这样，二进制数100101101100101就分解成4位一组的半字节，从右开始，就是100 1011 0110 0101。（请注意我偷懒了，我给你的数只有15位，所以有一个4位组是不全的。这个组总是在最左端，要用0来填补，所以你需要转换的真正值是0100 1011 0110 0101。）如果在表中查这四个值，你会发现它们对应于4、B、6及5。因此，相应的十六进制数就是0x4B65。

从相反方向，十六进制数0x18C3可转化为四个二进制数组0001（1）、1000（8）、1100（C）及0011（3），放在一起就得出0001100011000011。

类似的技术也适用于八进制（基数8），对应于表1-1的前两列并只使用最后3个（而不是4个）数字，如表1-3所示。采用这些表示方法和技术，你就能在一个足够大的寄存器中表示任

何非负的整数，并用我们讨论过的任何基数来解释。发挥一点创造性和才能，你甚至能使这些技术运用于象3或7这样怪异的基数。

表1-3 八进制↔二进制数转换

八进制	二进制	八进制	二进制
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

带符号表示

在真实世界中，常常使用负数。如果存储在寄存器中的最小可能值是0，那么，计算机如何存储负数值呢？这个相当奇怪的问题不是存储的问题而是解释的问题。虽然对于给定的寄存器大小可存储的最大模式数是固定的，但程序员可以将一些模式解释为负数值。通常的方法是采用一种称为二进制补码表示（two's complement notation）的解释方式。

通常人们认为，如果你倒车（甚至电影《春天不是读书天》（*Ferris Beuller's Day off*）中也这么认为），里程表会反转而且显然会让发动机取消几英里。我不知道是否会这样，对Ferris并非如此，但Howstuffworks.com[⊖]说应该这样。暂时想象会这样。假设我取来一辆相对较新的车（比方说，有了10英里的里程），并将其倒退行驶11英里。里程表会显示什么呢？

可是，里程表不会显示是"－1英里"。超过000, 000后，它可能会显示999, 999英里。但是，如果我再向前行驶1英里，里程表就又会转到000, 000。我们可以隐含地将－1定义为这样一个数字：当1加上它时，结果为0。

这就是二进制补码表示法的原理。负数是从全0的寄存器开始通过向反方向计数（采用二进制）来产生和处理的，如图1-10所示。首位是0的数被解释为正数（或0），而首位是1的数被解释为负数。

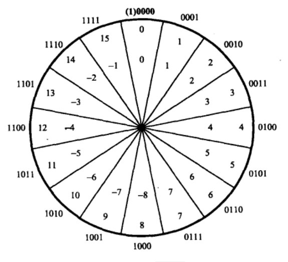


图1-10 4位带符号整数表示的结构

例如，数13写成8位二进制数为00001101（是十六进制数0x0D），而数－13是11110011（0xF3）。这些模式称为带符号（signed）数，以区别于先前定义的无符号（unsigned）数。特别是，模式0xF3是－13的二进制补码表示（在8位寄存器中）。

我们如何才能从0xF3得到－13呢？除了1打头，显然两种表示之间不存在相似性。两者之间的关系相当微妙，这是建立在上面所定义的负数是正数取反这一基础上的。就是说，13＋－13应该为0。采用上面的二进制表示，我们注意到：

⊖ http://auto.howstuffworks.com/odometer1.thm。

```
  00001101                 13
+ 11110011                -13  用二进制补码表示
------------
 100000000                  0  加上一个溢出/进位
```

然而，9位的量100000000（0x100）无法存储在一个8位寄存器中！就像当英里里程数变得过大时汽车里程表超过限度一样，8位寄存器也会溢出（overflow）并丢失包含在第九位的信息。这样就导致所存储的模式是00000000，即0x00，也就是二进制（十六进制也是）0。采用这个方法，我们就能看到，存储在8位寄存器中的值的范围是从－128（0x80）到＋127（0x7F）。大概一半值是正数，一半值是负数，这就是人们一般所想要的。图1-10显示，一般情况是，负数比正数多一个，因为图中0对面的数是负数。

这个展示非常依赖于使用8位寄存器。对于一个32位寄存器，就需要更大的值来产生溢出并循环回到0。－13的32位二进制补码表示不是0xF3，而是0xFFFFFFF3。事实上，0xF3若被看作是一个32位的数，则通常被解释成0x000000F3，它甚至不是负数，因为它的首位数字不是1。

手工计算一个数的二进制补码表示（对于固定的寄存器大小）并不困难。首先注意到，对于任何大小的寄存器，－1的表示都是寄存器所有的位均为1。向这个数加1将产生溢出并使寄存器的所有位均为0。对于任意给定的位模式，如果你对每个位进行反转（每个1变成0，每个0变成1，并保持原来的顺序。这个操作有时被称为按位（bitwise）取反(NOT)，因为它将NOT操作施加于每个位），并将得到的数再加上原来的数，得到的结果就是寄存器所有的位均为1。(为什么？）这个反转的模式（有时称为“1补码”或者称为“反码”）加到原来的模式，产生的和为－1。当然，再加上1就得到－1＋1，即0。这个反转的模式加1就得到原数的二进制补码。

```
  00001101   (= 13)
  11110010   (按位取反)
+        1
------------
  11110011   (= -13)
```

注意，再重复这个过程一次就使反转值再反转一次。

```
  11110011   (= -13)
  00001100   (按位取反)
+        1
------------
  00001101   (= 13)
```

这个过程可推广到任何数和任何（正数）寄存器大小。减法以同样方式进行，因为减去一个数就等同于加上这个数的负数。

浮点表示

除了表示有符号整数，还常常要求计算机表示小数或者带小数点的量。为做到这一点，可对标准的科学记数法进行修改，采用基于2的幂代替基于10的幂。这些数通常称为浮点(floating point）数，因为根据这种表示方式，它们包含可浮动的小数点。

从基本的数开始，显然任何整数都可通过增加一个小数点和很多个0的办法转换为带小数点的数。例如，整数5就变成5.000...，整数－22就变成－22.0000…，等等。对其他进制数也是这样（只不过从技术上讲小数（decimal）点针对十进制，而对其他基数则称为小数（radix）点）。这样，二进制数1010转换成小数就是1010.0000…，而十六进制数0x357转换成小数就是0x357.0000…。

任何带小数点的数都可通过移动小数点并乘以若干次基数而写成科学记数的形式。例如，阿伏加德罗数通常近似为$6.023 \cdot 10^{23}$。即便是忘记了它在化学中的重要性的学生也应能解释这种表示。阿伏加德罗数是一个24位（23+1）的数，其起始的四位数字是6，0，2，3，大约为602 300 000 000 000 000 000 000。科学记数法有三个部分：基数（base）（在本例中是10），指数（exponent）（23）及尾数（mantissa）（6.023）。为解释这个数，就要对基数求指数次的幂并与尾数乘。显然，有很多种不同的尾数/指数对能产生相同的数。阿伏加德罗数也可写成$6023 \cdot 10^{20}$、$60.23 \cdot 10^{22}$或者甚至是$0.6023 \cdot 10^{24}$。

计算机可采用同样的思想，但要用二进制表示。特别地要注意表1-4的模式。一个数乘以2就将其左移1位，除以2就将其右移1位。采用科学记数法形式也按同样的位模式移动（指数要适当调整），如表1-4所示。

表1-4 二进制数的指数表示

十进制数	二进制整数	二进制小数	科学记数法
5	00000101	00000101.0000...	$1.01(\text{binary}) \cdot 2^2$
10	00001010	00001010.0000...	$1.01(\text{binary}) \cdot 2^3$
20	00010100	00010100.0000...	$1.01(\text{binary}) \cdot 2^4$
40	00101000	00101000.0000...	$1.01(\text{binary}) \cdot 2^5$

我们将这种方法进行扩展来表示二进制的非整数浮点数，如表1-5所示。例如，数2.5是5的一半，可表示成二进制量10.1或者二进制量1.01乘以2^1。那么，1.25就是二进制1.01或者1.01乘以2^0。用这种表示，任何十进制浮点量都有一个等价的二进制表示。

表1-5 二进制小数的指数表示

十进制数	二进制数	二进制实数	科学记数法
5	101	00101.000...	$1.0100(\text{binary}) \cdot 2^2$
2.5		00010.100...	$1.0100(\text{binary}) \cdot 2^1$
1.25		00001.010...	$1.0100(\text{binary}) \cdot 2^0$
0.6125		00000.101..	$1.0100(\text{binary}) \cdot 2^{-1}$

电气和电子工程师协会（IEEE）已经发布了一系列的规范文献，描述了用二进制表示浮点数的标准方式。其中一个标准IEEE 754－1985就描述了将浮点数存储成32位字的方法，如下所示（如图1-11所示）：

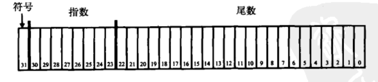


图1-11 IEEE 32位浮点存储：符号、指数、尾数

对字的32位编号，最左端为第31位，最右端是第0位。首位（第31位）是符号（sign）位，表明这个数是正数还是负数。与二进制补码的表示不同，这个符号位是能显示量值相同的两个不同数的唯一方式。

再往后的8个位（第30－23位）是“偏置的”指数（exponent）。要是用位模式00000000来代表2^0，用位模式00000001来代表2^1是可以的，但是，那样的话我们就不能表示象0.125（2^{-3}）

这样的小数值。用8位二进制补码表示也是有意义的，但IEEE也没有这样选择。IEEE选择的是采用无符号数0...255，但存储在指数位（表现指数）的数是一个“偏置的”指数，实际值比“真实”指数大127。换句话说，一个实际为0的指数被存储为表现指数127（二进制01111111）。一个实际为1的指数将被存储为表现指数128（二进制10000000），而存储成00000000的指数实际代表的是很小的量2^{-127}。其余的23位（第22－0位）是带小数点(decimal point)（技术上称为小数点（radix point)，因为我们再也不用处理十进制了）的尾数，传统上被放置在紧跟第一位二进制数的后面。这个格式在小数点前面有固定位数，有时称为范式（normalized form)。

这样，对于常规的数，存储在寄存器中的值就是：

$$(-1)^{符号位} \cdot 尾数 \cdot 2^{真实指数+127}$$

表现指数为127就意味着尾数乘以1（相应的真实指数是0。所以乘数是2^0)，而表现指数为126就意味着小数乘以2^{-1}即0.5，依此类推。

实际上，上式中有一个小小的陷阱。因为数是二进制表示的，第1个非零数字必须是1（如果不是0没有其他的选择！）。由于知道第一位数字是1，我们就能将其略去以便利用节省下来的空间来存储我们预先不知道的其他数字。所以，真实公式应该是：

$$(-1)^{符号位} \cdot 1.尾数 \cdot 2^{真实指数+127}$$

作为一个简单的例子，十进制数2.0用二进制表示是$1.0 \cdot 2^1$。将这个数表示成一个IEEE浮点数，符号位是0（是一个正数），尾数全是0（还有一个隐含的首位1)，指数是127＋1，即128，也就是二进制数10000000。这将会以32位的量存储。

```
0      10000000    00000000000000000000000
符号    指数                 尾数
```

这个位模式可写成十六进制数0x40000000。

另一方面，数－1.0的符号位为1，指数为127＋0，即二进制数01111111，尾数全是0（加上一个隐含的首位1)。这将产生下面的32位量：

```
1      01111111    00000000000000000000000
符号    指数                 尾数
```

这个位模式的另一种写法是0xBF800000。

这里还有个小问题。如果要存储的数就是0.000...你会怎么办?在一串0中没有任何地方放置“隐含的首位1”。作为一个特殊情况，IEEE定义所有位全是0（符号、指数、尾数）的位模式表示数0.0。还有一些其他的特殊情况，包括正和负的无穷大，还有所谓的NaN（不是一个数，这是当你试图做一个非法操作如对负数求对数值时所产生的结果)，以及用来以极大的精确度表示极小数（但很少使用）的“非规范化数”。IEEE还为在64位和更大的寄存器中更准确地存储数定义了标准方法。对于上面描述的32位标准，这些额外的情况尽管在细节上不同，在本质上是类似的。但细节相当枯燥和技术化，不要求一般程序员掌握。

在任何基于基数的表示中都会出现一个问题，会有一些不能准确表示的数。即使不用担心无理数（如π)，一些分数也不能准确表示。例如，在基数为10时，分数1/7有近似值0.14285714285714...但永远不会结束。分数1/3也类似，它的值为0.33333...。在基数为2时，分数1/3是0.0101010101010101...但无法将一个无穷的序列放入到23个尾数位中。所以，解决方法就是：我们不这样做。

我们采取的方法是尽可能地接近原来的值。转换成小数后，我们看到1/3大概等于

1.01010101010101010101010 · 2^{-2}，表示成：

```
0    01111101  01010101010101010101010
符号    指数              尾数
```

这不是一个完美的表示，但它是接近的。“接近”可能并非在所有情况下都足够好。小的误差（称为舍入误差（roundoff error））将不可避免地进入到涉及浮点数的计算中。如果将这个数乘以3（1.1 · 2^1），不太可能得到准确的1.0这个值。程序员们，尤其是那些处理矩阵求逆之类大数值计算的程序员们，要花费很多时间来努力使这种误差最小化。现在，我们能做的就是要知道有这个问题。

对浮点数执行操作是棘手的（而且常常很慢）。直观上看，所采用的算法是可以理解的，并且非常类似于你已知道的科学计数法表示的数的操作。乘法相对容易，因为我们知道2^x乘以2^y就是2^{x+y}。因此，对两个浮点数求乘积，结果的符号位就是两个数的符号位的积或者异或（XOR），尾数就是两个数的尾数的乘积，而指数就是两个数指数的和⊖。这样，我们就有下面的例子：

```
3.0 = (0) (10000000) (10000000000000000000000) [0x40400000]
2.5 = (0) (10000000) (01000000000000000000000) [0x40200000]
```

3.0 · 2.5的结果的符号位就是0，尾数就是1.100... · 1.010...，即1.111...（你明白为什么吗？）。最后，指数就是1＋1＝2，表示成1000001。这样，乘积就是：

```
(0) (10000001) (11100000000000000000000) [0x40F00000]
```

正如所预期的，这个数转换后就是7.5。

加法要困难得多，因为只有两个数的指数相同时加法才能进行。如果两个相加的数的指数相同，则将其尾数相加就（几乎）足够了：

```
3.0 = (0) (10000000) (10000000000000000000000) [0x40400000]
2.5 = (0) (10000000) (01000000000000000000000) [0x40200000]
```

二进制数1.01加上1.1得10.11，所以答案就是10.11乘以共同的指数部分：10.11 · 2^2。得到的结果是：

```
(0) (10000001) (01100000000000000000000) [0x40B00000]
```

转换后，正如预期，得到结果5.5。

然而，当两个指数不相同时（例如，当将2.5和7.5相加时），就要将一个加数转换成与另一个加数的指数相同的等价形式：

```
2.5 = (0) (10000000) (01000000000000000000000) [0x40200000]
7.5 = (0) (10000001) (11100000000000000000000) [0x40F00000]
```

下面来转换7.5：1.111 · 2^2与11.11 · 2^1等价。11.11+1.01得101.00。101.00 · 2^1等价于1.01 · 2^3。最终结果就是：

```
(0) (10000010) (01000000000000000000000) [0x41200000]
```

也正如所预期的，这个值是10.0。

字符表示

非数值数据如字符和字符串也按二进制数来处理，只是程序员/用户对它们的解释不同。存储字符最常用的标准是ASCII码，字面意思是美国信息交换标准码（American Standard

⊖ 但记住，要考虑到指数的偏差以及尾数头部上没有表示出来的那个1。

Code for Information Interchange)。ASCII码将特定字符分配到0到127之间的各个数。

这有点过于简化了。从技术上说，ASCII码为0000000到1111111之间（包括这两个数）的每个7位二进制模式提供了一个解释。其中的很多模式被解释成字符。例如，模式1000001是一个大写的‘A’。一些二进制串，尤其是0000000到0011111之间的那些串被解释成“控制字符”，如回车，或者到外设的命令如“标题开始”或“传输结束”。由于几乎所有的计算机都是面向字节的，所以多数计算机不是将ASCII字符存储成7位的模式，而是存储成8位的模式，首位是1。例如，字母A就是二进制01000001，也就是十六进制0x41，或者说是十进制65。采用8位存储就使得计算机能利用额外的（高位）模式进行字符集扩展。例如，在微软的Windows中，几乎每个字符集都有128－255范围内的不同显示值。这些显示值可能包括图形字符、扑克牌花色、带区分标记的外文字符，等等。

ASCII编码的主要优势是每个字符都很宽裕地对应一个字节。ASCII的主要缺点是：由于它是美国的标准码，所以不能很好地反映世界范围内字母表和字母的变化。随着因特网不断地将不同国家和讲不同语言的人们连接在一起，显然就需要有对非英语（或者说，非美国）字符的编码方法。这就出现了Unicode共同体发布的UTF-16编码。

UTF-16采用两个字节（16位）来存储每个字符。前128个模式几乎与ASCII码一样。然而，由于有16位，就有超过65 000（在技术上是65 536）个不同的模式，每个模式可分配一个单独的（可打印）字符。这个巨大的字符集合就使得对用多种字母表写成的文档进行统一、可移植的方式处理成为可能，这些字母表包括：（美国）英语，不常见的字符如音标（如æ）和货币符号，拉丁字母表的变型如法语和德语，以及（从美国计算机科学家的角度看）不常见的字母表如希腊语、希伯来语、西里尔语（用于俄语）、泰语、切罗基语以及西藏语。例如，希腊语大写字母psi（ψ）就是用0x803A，即二进制数1000000000111010来表示。即使是对汉语/日语/朝鲜语这样的象形文字集合（包含的字符超过40 000个）也能表示（见图1-12的例子）。

机器操作表示

除了存储很多不同类型的数据，计算机还需要存储可执行的程序代码。如同我们讨论过的所有其他模式一样，在最基本的级别上，计算机只能解释二进制的活动模式。这些模式通常称为机器语言（machine language）指令。寄存器在CPU中的主要角色之一就是取出并保存一个位模式，这个位模式可被译码成为机器指令然后执行。

机器语言的解释一般是困难的并且在计算机之间变化极大。对于任意给定的计算机，指令集（instruction set）定义了计算机能执行哪些操作。例如，Java虚拟机（JVM）有一个相对小的指令集，只有256种可能的操作。每个字节就可解释成要采取的可能动作。值89（0x89）对应于dup指令，该指令导致计算机对存储在CPU中的特定部分信息进行复制。值146（0x92）对应于i2c指令，该指令将一个32位的量（通常是整数）转换成一个16位的量（通常是一个统一编码的字符）。这些数字到指令的对应关系特定于JVM，对奔腾4或PowerPC是不适用的，这两种CPU有其特定的指令集。

为完成某个任务而生成机器码常常是极为费力的工作。计算机通常为该任务提供大强度的编程支持。程序员通常用某种人可读的语言写程序，然后采用如编译器这样的程序将该语言转换成机器码。编译器本质上就是将人可读的语言转换成机器语言的程序。

解释

根据我们在前面部分所学到的知识，显然任何位模式都几乎能以几种不同的方式进行解释。一个给定的32位模式可能是一个浮点数、两个UTP-16字符、几条机器指令、两个16位整数、一个无符号32位整数或很多其他可能性（见图1-12）。计算机如何区分两个相同的位串呢？

解释为	0011 1110	0010 0000	0111 0010	0011 0111
十六进制数	3E	20	72	37
带符号32位整数	1042313783			
带符号16位整数	15904		29239	
带符号32位浮点数	0.156686			
8位字符	>	space	r	7
16位字符	㸠		爷	
JVM机器指令	istore_3	lstore_2	frem	lstore

图1-12 位模式的多种解释

简单的答案是它不能区分，这会存在误导。较长但更有用的答案是对位串的区分是由程序指令的上下文来提供的。我们将在接下来的章节中讨论到，多数计算机（包括已讨论过的主要计算机JVM）都有若干不同种类的指令做着大致相同的事情。例如，JVM就有不同的指令来完成32位整数、64位整数、32位浮点数、64位浮点数的加法操作。这些指令隐含认为要相加的位模式就是适当的类型。如果你的程序装入两个整数到寄存器中，然后告诉计算机做两个浮点数的加法操作，计算机就会天真而无辜地将两个整数位模式当作浮点数处理，将它们相加，得到无意义而且完全错误的结果。（图1-12表明了这一点，因为该模式中所隐含的指令在类型上是非法的。）类似地，如果你告诉一个计算机将一个浮点数作为一个机器代码来执行，计算机就会尽最大能力地来执行，不管这个数对应了多么愚蠢的指令。如果你幸运的话，这只不过会使你的程序崩溃。如果你不走运，...那么，这就是黑客侵入计算机的主要途径之一：通过使缓冲器溢出并用他们自己的指令覆盖可执行代码。

某种类型被看作另外不同的类型来使用几乎总是会出错。不幸的是，这种错误计算机只能部分地弥补。最终，确保数据正确存储和位模式正确解释要由程序员（以及编译器编写者）负责。JVM的主要优势之一就是它能捕获这些错误。

1.3 虚拟机

1.3.1 什么是虚拟机

由于指令集之间的差异，那么，就很有可能（提醒：这是一个相当保守的说法）使得为一个特定的平台（CPU类型和/或操作系统）所写的程序不能运行在不同的平台上。这就是为什么软件发行商为使用奔腾4 CPU的Windows计算机和使用PowerPC CPU的Macintosh计算机卖出不同版本的程序，也就是为什么很多程序有“最低配置”，即某个特定的计算机必须有一定的存储容量或某种特别的视频卡才能正常工作。在极端情况下，这就要求计算机程序员独立地从头开始写相关的程序（比如同一个游戏的Mac和Windows版本），这实质上是一个使程序开发的时间、精力以及成本都加倍的过程。幸运的是，很少需要如此。多数的编程是用所谓的高级语言（high-level language）如C、C++、或Java来完成的，然后人可读的程序源码再由另外一个程序如编译器转换成可执行的机器码。只有少部分的程序——如嵌入式系统、需要直接硬件访问的图形功能或者控制非一般外设的设备驱动器，才需要用机器特定的语言来编写。

Java的设计者意识到了Web的普及性以及需要在任何地方都能运行可编程Web页面，因此采取了不同的途径。Java本身是一个高级语言。Java程序一般编译成类文件（class file），每个文件对应于程序或程序一部分的机器语言。与从C、Pascal、或C++编译而成的标准可执行

码不同，类文件无需对应于编写和运行程序的实际机器。类文件用机器语言编写，采用Java虚拟机（Java Virtual Machine, JVM）的指令集，JVM只作为一个软件仿真，它是一个模仿芯片的计算机程序。这个“机器”有正常物理计算机（如Intel奔腾4）典型的、有时甚至更强大的结构和计算能力，但却没有物理芯片的很多问题和局限性。

JVM通常是一个运行于宿主机上的程序。像任何其他可执行程序一样，它采用本地机器的指令集运行于物理芯片上。这个程序有一个特殊的用途：其主要功能是解释和执行用JVM机器语言写的类文件。通过运行一个特定的程序，安装在计算机中的物理芯片就可扮作JVM芯片，因而就能运行用JVM指令集和机器码写成的程序。

“虚拟机”的思想并非新生事物。1964年，IBM开始着手研发VM/CMS，这是一个面向System/360的操作系统，能为许多用户同时提供分时服务。为了将计算机的服务全部地提供给每个用户，IBM的工程师们决定建立一个软件系统和用户接口，给用户这样一个感觉：他或她自己在享有整个计算机的全部服务。每个人或程序都能按需支配整个虚拟S/360而不用担心该程序是否会导致另一个人的程序崩溃。这也使得工程师们能大幅度地更新或改进硬件而不用强迫用户重新学习或重写程序。二十多年以后，VM/CMS还在大型IBM主机上使用，运行为虚拟S/360而设计和编写的程序，运行虚拟S/360的硬件已经快了几乎几百万倍。其后，虚拟机和仿真器已经成为很多编程工具和语言（包括Smalltalk，一种早期的面向对象的语言）的标准件。

补充资料

.NET框架

另一个常用虚拟机的例子是.NET框架，由微软开发并在2002年发布。.NET框架是Visual Studio当前版本的基础，并为很多基于网络的技术如ASP.NET、ADO.NET、SOAP以及.NET Enterprise Servers提供了一个统一的编程模型。仿效Sun的JVM先例，.NET引入了一个通用语言运行时（Common Language Runtime，CLR），这是一个虚拟机，能管理和运行用若干种语言开发的代码。该虚拟机采用了虚拟指令集，称为微软中间语言（Microsoft Intermediate Language, MSIL），在思想上非常类似于JVM的字节码。甚至CLR执行引擎的详细体系结构也类似于JVM。例如，它们都是基于堆栈的体系结构，并为面向对象的、基于类的环境提供指令集支持。像JVM一样，CLR的设计具有虚拟机的大多数优势：抽象可移植的代码可广泛地分布和运行而无需牺牲本地机器的安全性。微软还仿效Sun开发和发行了一个巨大的预定义类库，为使用.NET框架并且不愿意从头开发类的程序员提供支持。两个系统的设计都考虑了Web服务和移动应用。与JVM不同的是，MSIL或多或少是从头开发的，以支持多种编程语言，包括J#、C#、Managed C++及Visual Basic。微软还建立了一个标准的汇编器Ilasm。实际上，CLR作为一个计算环境没有JVM那样通用和广泛发布，但是软件市场变化多端，最终结果尚无定论。

一个可能会有重要影响的关键问题是微软提供多平台支持的程度。在理论上，MSIL与JVM一样是与平台独立的。但是，.NET框架库的相当大的部分基于较早的微软技术，不能成功地运行于Unix操作系统上，甚至不能运行于较早的Windows系统上（如Windows98）。微软支持非微软（或者更早微软）操作系统的历史记录未能鼓励第三方开发者来开发跨平台的MSIL软件。对比之下，Java已经建立了一个包括所有平台的强大的开发组，依赖并支持这种可移植性。如果微软能继续履行其多平台支持的承诺，并开发出足够的客户基础，.NET也许能取代Java作为开发和部署可移植的基于Web应用的首选系统。

1.3.2 可移植性问题

虚拟机（特别是JVM）的一个主要优势是，只要有适合的解释器，就可在任何地方运行。与Mac程序需要PowerPC G4芯片才能运行（也有G4仿真器，但很难找到而且很昂贵）不同，JVM对于几乎所有计算机和很多种设备如PDA都可用。每个主要的Web浏览器如Internet Explore、Netscape或Konqueror都有一个JVM（有时称为“Java运行时系统（Java runtime system)”，建立该系统以使Java程序正确运行）。

进一步地，JVM能在任何地方连续运行，因为程序本身受基础硬件变化的影响相对较小。一个Java程序（或者Java类文件），除非考虑速度，它在一个为旧式奔腾计算机所写的Java仿真器上运行与在为顶端Power G7（一款新的尚未出现的机型）所写的Java仿真器上运行会有相同的表现。如果/当Motorola开发出产品时，几乎一定会有人为它写一个JVM客户程序。

1.3.3 超越限制

虚拟机（特别是JVM）能提供的另一个优势是超越、忽略或屏蔽基础硬件所施加限制的能力。JVM有虚拟部件，对应于上面讨论的真实计算机的部件，但因为它包含的只是软件，所以制造成本微乎其微。因此，这些部件的大小和数量均可按程序员的需要设置。芯片上每个实际的寄存器都要占据空间、消耗功率，并花费大量的金钱，因此，寄存器常常是相当短缺的。在JVM上，寄存器实际上是免费的，程序员可任意使用。连接CPU到存储器的系统总线的大小也可按程序员的需求设置。

历史上的真实例子有助于说明这个道理。初始的IBM-PC是基于Intel 8088芯片的。8088又基于Intel另外一款更早的芯片8086，两款芯片在设计上几乎相同，但8088采用的是8位总线而不是8086的16位总线。这无疑限制了8088的CPU和存储器之间的数据传输速度，相对于8086减小了大约50%，但IBM决定选择8088从而将制造成本和售价保持在低水平。不幸的是，这个决定在接下来的15年左右的时间限制了PC的性能，因为IBM、Intel和微软被要求在所谓的Intel 80x86系列的每个后代产品中都保持向后兼容。只有Windows的开发才使微软最终能充分利用较低的制造成本和更宽的总线。随着多数主要的软件和硬件制造商竞相在它们的产品中为若干种不同的芯片和芯片集提供支持，在高端芯片集中可利用的某些特性在低端却不能得到，类似的问题还在发生。适当编写的Java运行时系统能利用（专门为新的体系结构所写的JVM上）可得到的特性，并使其为所有Java程序或JVM类文件所用。

与多数实际的计算机芯片相比，JVM的优势还在于其设计在本质上更好、更整洁。部分原因是它是在1995年从头开始设计的，而不是通过继承若干代较早版本的工程折衷而得到的。设计者不必担心像芯片大小、功耗或成本这样的物理限制，由此而导致的简单性意味着设计者能够将其注意力关注于如何生成在数学上易处理和优雅的设计，并使高层次特性的加入成为可能。特别是JVM还考虑到更高程度的安全增强技术，这一点我们将在下面简单说明，并在第10章详细讨论。

1.3.4 易于升级

与升级硬件的困难性相比，虚拟机的另一个优势是易于升级或改变。有大量证据证明，1994年发布的一款奔腾芯片有一个错误，即奔腾P54C的FPU不能正确工作。不幸的是，纠正这个缺陷要求消费者将旧芯片发回到Intel并进行完全的物理替换。与此形成对照的是，JVM实现上的故障可由程序员在软件中修复，甚至可以由能进行源码访问的第三方通过正常渠道分发，包括简单地在因特网上提供。

类似地，一个新的改进JVM版本，或许算法得到更新或者速度得到大幅度提高，就可像

任何其他升级的程序（如视频卡驱动器或对标准程序进行安全升级）一样容易地发布。由于JVM只是软件，多疑的用户甚至能保持连续若干个版本，在新版本有某些微妙和未发现的错误时，她还能返回到旧版本以使程序仍能运行。（当然，你只在发现该用户是正确的之前才认为她是多疑的，在这之后你会认识到她只是谨慎和负责任。）

1.3.5 安全问题

虚拟机的最后一个优点是，在基础硬件的协作下，可进行配置而运行在更安全的环境下。Java语言和JVM的设计特别考虑了这种对安全性的增强。例如，多数的Java程序不需要访问宿主计算机的硬盘，若提供这种访问，尤其是写硬盘的能力，可能会使计算机面临计算机病毒的感染、数据失窃或者直接将极重要的文件删除或毁坏等危险。虚拟机由于只是软件，故能详查磁盘的访问企图，并实施比操作系统本身所能实施的更为精密的安全策略。

JVM甚至走得更远，其设计不仅为了安全性也为了某种程度的可验证安全性。程序中的很多安全缺陷是意外产生的，比如，程序员试图读一个串却没有确保已分配足够的空间来保存它，甚至试图执行一个非法的操作（或许是将一个数用ASCII串来除），结果就导致了不可预测和可能有害的结果。JVM的字节码被设计成可验证的，这种验证是通过采用计算机程序检验这种意外错误来实现的。这不仅降低了出现有害安全缺陷的可能性，也改进了软件的总体质量和可靠性。软件错误毕竟不一定会摧毁系统和允许非授权访问。很多错误会默默潜伏着（几乎是检测不到），然后产生错误的应答。在产生错误应答之前，有时甚至是在程序运行之前捕获这些错误将会大幅度地提高程序的可靠性。JVM安全策略将在第10章进行广泛的讨论。

1.3.6 劣势

如果虚拟机是这么的神奇，为什么它们没有更普及呢？主要的原因，用一句话说，就是“速度”。进行一个特定的操作，通常用软件完成要比硬件完成少花费1000倍的时间，所以用Java在JVM上做核心的计算（矩阵相乘、DNA测序等等）比用从C++编译得到的（面向特定芯片的）机器语言执行同样的计算要慢得多。

幸运的是，实际的差距看起来没有近乎1000倍那么大，主要的原因是已有一些非常聪明的JVM实现者。一个写得好的JVM会尽可能利用可得到的硬件做尽可能多的操作。这样，程序就会（例如）利用本地机器的电路做加法，而不是全部用软件来模拟。编译技术的进步已使得Java运行速度几乎与本地编译的代码一样快，通常在2倍运行时间之内，有时几乎感觉不出Java较慢。*JavaWorld*在1998年2月的一项研究（“性能测试显示Java和C++一样快”）发现：对于一组测试基准任务，带有高效编译器的高性能JVM所生成的程序一般在最差时也仅仅比对等的C++程序运行得稍慢（大约5.6%），而在多数情况下已到了度量的极限。当然，比较计算机和语言的速度是极为困难的过程，就像将标准的苹果和桔子做比较一样，不同的研究者会发现不同的价值。

在某些方面，速度可能不成问题。对于非大量数值计算的大多数用途，Java或任何其他合理的语言已经快得足够满足多数人的需要。Java及其扩展JVM提供了一组强大的基本计算程序，包括广泛的安全度量，这些都是在如C++等很多其他语言中没有的。没有理由认为，导致计算机快速崩溃与虽有点慢但能够运行完程序相比会是一个巨大的优势。

一个更重大的劣势是在程序员和实际物理硬件之间插入了JVM解释器。对很多要求汇编语言编程（游戏、高速网络接口或配备的新外设）的应用，采用汇编语言的关键原因是能够在很低的层次上对硬件进行直接的控制（尤其对于外设），通常是按位和按线来控制的。一个写得不好的JVM阻碍了这种直接控制。随着对JVM的硅实现的开发，这越来越不成问题，比

如aJile Systems的aJ-100微控制器或Zucotto Systems的Xpresso系列的处理器。两个公司都在生产适合于手持的因特网设备的控制器芯片，并将基于芯片的JVM字节码的实现作为机器码。换句话说，这些芯片不要求任何软件支持来将JVM字节码转换为本地指令集，因此能以全硬件速度运行，并在位的层次上对硬件实施全面的控制。随着Java芯片的开发，JVM已经完成了一个全面的转变，这个转变是从允许对各种不同处理器进行可移植和安全访问的精巧的数学抽象开始，到其自身发展成为物理芯片。

1.4 JVM编程

1.4.1 Java：JVM不是什么

必须强调，Java不等同于Java虚拟机，虽然它们是一起被设计出来的，并且常常一起使用。Java编程语言是一种高级编程语言，它设计成可支持在诸如因特网的分布式网络环境中的安全且平台无关的应用。也许Java最常见的使用方式是创建“applets”，作为网页的一部分下载并与浏览器交互作用而无须占用服务器的网络连接。与此不同，Java虚拟机是一个共享的虚拟计算机，它为Java应用的运行提供了基础。

Java设计的很多方面强烈地影响了JVM。例如，JVM是一个虚拟机恰是因为Java应该是一个平台无关的语言，所以它不能对观察者使用什么类型的计算机作出任何假设（见图1-13）。JVM是围绕安全验证器的观念设计的，所以Java程序能运行在安全环境中。网络支持是在相对较低的层次上建立在JVM标准库中，以确保Java程序能够使用一个标准化的、有用的网络操作集。然而，两个产品之间没有必然的关联。Java编译器在理论上能编译得到PowerPC（较早的Macintosh计算机的基础硬件）的本地代码而不是编译为JVM字节码，而任何其他语言的编译器也可编译得到JVM代码。

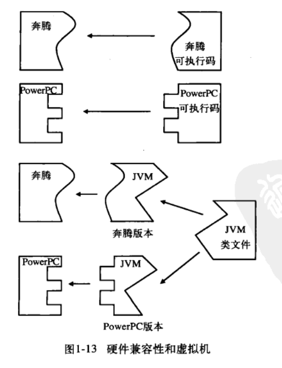


图1-13 硬件兼容性和虚拟机

特别地，请见图1-14中的程序。这是一个简单的Java程序示例，其功能是输出一个给定的串到缺省的系统输出，通常是当前活动窗口。这个程序，或者类似的程序，常常是任何初级编程课程的第一个标准示例，虽然有时程序也可能是打开一个窗口并显示一个消息。然而，当更详细地考察时，却发现Java程序本身没做任何这类事情。为了得到执行，它必须首先通过某种转换程序（编译器）来生成一个可执行的类文件。只有这个类文件才能运行以产生期望的输出。

Java或任何编程语言，更适于被看作是用于指定计算机所执行计算的一种结构。用任何语言写的程序都是对一种特定计算的规范。为了使计算机完成工作，这种规范（程序）必须被转换成机器码，计算机最终将其辨析为一个程序步骤序列。

```
public class JavaExample {
        public static void main(String [] args) {
                System.out.println("This is a sample program.");
        }
}
```

图1-14　Java程序样例

1.4.2　样例程序的转换

通常有很多方式来指定一个给定的计算。可用很多其他语言写出一个非常类似的程序。例如，图1-15、1-16、1-17显示了用C、Pascal及C++写出的具有相同行为的程序。这些程序在总体结构上也显示出极大的相似性。程序的核心是一行代码，以某种方式访问一个固定的串（由ASCII字符组成）并调用一个标准库函数将其传递到缺省的输出设备。其差别是微妙的并涉及细节。例如，Pascal程序有一个名字（PascalExample），而C和C++版本没有。在C和C++中，程序必须都显式地表示行末回车，而Java和Pascal有一个功能，可自动在行末加一个回车。然而，这些差异与显著的相似性相比就相当小了，尤其是考虑这与机器码之间的差异时。

```
#include <stdio.h>

int main()
{
        (void) printf("This is a sample program.\n");
}
```

图1-15　C程序样例

```
Program PascalExample(output);

begin
        writeln ('This is a sample program.');
end.
```

图1-16　Pascal程序样例

```
using namespace std;
#include <iostream.h>
int main()
{
        cout << "This is a sample program." << endl;
}
```

图1-17　C++程序样例

根据前面对典型计算机的体系结构和组织的讨论，考虑以下问题：很少CPU（如果有的话）在ALU或控制器内有足够的寄存器来存储整个串。（的确如此，因为在抽象中对串的长度没有一个必要的上界。否则的话，计算机就不会只输出一个句子，而是输出整个课本了。）没有CPU能用单个指令将串输出到屏幕。代之的是，编译器必须将程序分解成足够小的步骤，这些步骤都在计算机的指令集之内。

1) 串本身必须存储在计算机存储器中的某个地方；
2) CPU必须确定存储位置；
3) CPU还必须确定哪个外设输出消息；
4) 必须确定消息的类型是什么；
5) 必须将适当的指令传递到外设；
6) 告诉外设串的存储位置；
7) 告诉外设该串是一个串（不是一个整数或是一个浮点数）；
8) 告诉外设做适当的动作将串输出（可能还要自动加上一个回车）。

一行代码可能要转换成8个或更多的计算机必须执行的单个操作。事实上，单行代码中所包含的要执行的操作数量没有限制。像如下的Java代码

```
i = 1+2+3+4+5+6+7+8+9+10;
```

对每个加法都需要一个操作，故在此情况下就有9个单独的计算。由于对数学公式的理论复杂度没有限制，因此对单个Java语句的复杂度也没有限制。

1.4.3 高级语言和低级语言

将很多机器语言指令封装到单行代码是高级（high-level）语言的典型特征。Java、Pascal等是这种语言的典型例子。Java编译器的任务就是接受复杂的语句或表达式并生成适合的机器语言序列来执行这个任务。

与此形成对照的是，低级（low-level）是以机器语言中的操作与程序代码中的语句之间有非常紧密的关系为特征的。采用这个定义，机器语言当然是一种非常低级的语言，因为一个机器语言程序与其自身之间总有一个1对1的关系。汇编语言（assembly language）是一个对人而言稍微易读的语言，但仍是低层语言，其设计是为了提升对机器及机器代码指令的总体控制，但仍保证程序员可读并理解。汇编语言也以汇编语言指令与机器代码指令之间的1对1关系为特征。

在机器语言中，指令的每个要素（也称为操作码opcode，是operation code的简写）像计算机中的其他任何内容一样是一个数。前一节提到操作码89（0x59）对JVM是有意义的，它将导致JVM复制一部分信息。在汇编语言中，对每个操作码都给出一个助记符（mnemonic）（读做“num-ON-ik”，来自希腊文字，是记忆的意思），它解释或概述了该指令确切要做的工作。与操作码89对应的助记符是dup，是“duplicate”的简写。类似地，助记符iadd是“integer add”的简写，对应于执行整数加法的机器码。转换程序在这里称为汇编器（assembler），其任务是将每个助记符转换成适当的二进制操作码的形式，并收集计算机所需要的数据（见图1-18）。

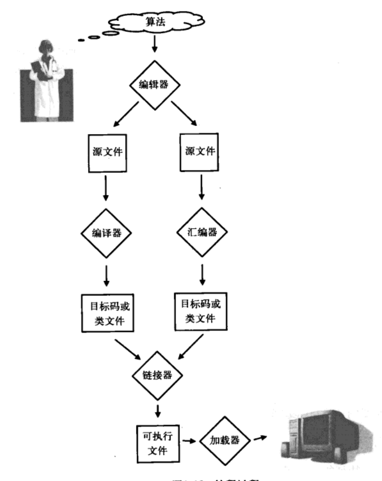


图1-18　编程过程

1.4.4　JVM所看到的样例程序

上面样例程序的JVM机器码版本将会是很长且大部分不可读的二进制串。图1-19显示出与机器码对应的程序版本。该程序用低层次JVM汇编语言jasmin手工写成，但也可根据原始的样例程序通过一个编译器生成。注意程序要长许多，几乎是30行而不是只有3行或4行，并且更难于理解。这是因为在写高级语言程序时你认为理所当然的很多事情在这里就必须精确和明白地予以指明。例如，在Java中，除非另外指定，每个类隐含地都是一个Object类型的子类。而JVM则要求明确地定义每个类与其他类的关系。图1-19中的注释（以分号;开头）对于如何精确地定义和完成（隐含）操作给出了更为详细的、逐行的描述（面向人类阅读者）。

要注意，虽然在Java中明确地支持和使用类、子类等概念，但在本节中给出的JVM低级语言程序中并没有特别针对Java的内容。事实上，就像Java编译器的任务是将高级Java代码转换成类似于JVM的机器码一样，C++或Pascal编译器的任务是对特定平台的程序代码做同样的事情。没有理由认为Pascal编译器不能设计成生成JVM码而不是PowerPC或奔腾机器码作为最终（编译）输出。程序然后就运行于一个JVM上（例如，运行在一个Web浏览器中的JVM仿真器上，或者像较早提到的专用芯片上），而不是特定的芯片硬件上（见图1-20）。

```
; 定义相关的类文件为 jasminExample.class
.class public jasminExample
; 定义jasminExample为Object的子类
.super java/lang/Object

; 创建对象所需的标准步骤
.method public <init>()V
     aload_0

     invokespecial java/lang/Object/<init>()V
     return
.end method

.method public static main([Ljava/lang/String;)V
     ; 对于System.out和字符串，我们需要两个堆栈元素
     .limit stack 2

     ; 寻找System.out（一个PrintStream类型的对象）
     ; 并将其放入堆栈
     getstatic java/lang/System/out Ljava/io/PrintStream;

     ; 寻找要打印的串（字符）
     ; 并将其放入堆栈
     ldc "This is a sample program."

     ; 调用PrintStream/println方法
     invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

     ; ...大功告成!
     return
.end method
```

图1-19 JVM汇编语言的样例程序

Java码	Jasmin码	JVM机器码
$x=1+2+3+4+5;$	bipush 1	0x10
		0x01
	bipush 2	0x10
		0x02
	iadd	0x60
	bipush 3	0x10
		0x03
	iadd	0x60
	bipush 4	0x10
		0x04
	iadd	0x60
	bipush 5	0x10
		0x05
	iadd	0x60
	istore_1	0x3c

图1-20 Java、jasmin以及JVM机器码中的样例表达式（细节见下一章）

1.5 本章回顾

- 计算机只是高速执行算法的设备，其作为电子设备的构造细节并没有其作为信息处理设备的工作细节那样重要。
- 计算机的主要部件是中央处理单元（CPU），CPU又包含控制单元、算术和逻辑单元（ALU）、以及浮点单元（FPU）。这就是所有计算实际发生的地方和程序实际运行的地方。
- 存储器和外设通过一组总线连接到CPU，总线携带到达CPU和来自CPU的信息。
- 所有存储在传统计算机中的值都以二进制数字或位的方式存储。为方便起见，这些位组成较大的单位如字节和字。一个特定的位模式可能有若干种解释，如一个整数、一个浮点数、一个字符或一序列字符，甚至计算机本身的一组指令。确定一个给定的位模式代表哪种类型的数据取决于上下文并在很大程度上是程序员的责任。
- 不同的CPU芯片有不同的指令集，代表CPU能做的不同事情和做事情的方式。每个CPU需要以不同机器语言写成的可执行程序，即使运行相同的程序。
- 虚拟机是一个程序，运行于真实CPU之上，并解释机器指令就好像它本身是一个CPU芯片。Java虚拟机（JVM）就是这种程序的一个常见例子，在几乎每台计算机和每个Web浏览器上都可能找到。
- 虚拟机比传统基于硅的CPU有很多优势，如可移植性、很少的硬件限制、易于升级以及安全性。虚拟机可能在速度上有很大劣势。
- Java是能将很多机器码指令结合成单个语句的高级语言的例子。C、Pascal和C++是类似的高级语言。这些语言必须进行编译，将其代码转换成机器可执行的格式。
- 低级语言如汇编语言在程序语句与机器指令之间有一个紧密的、通常是1对1的关系。
- Java与JVM之间没有必然的联系。多数Java编译器能编译到JVM可执行码，但是C、C++编译器也能做到这一点。类似地，一个Java编译器能编译到奔腾4本地码，但是所产生的程序不能在Macintosh计算机上运行。

1.6 习题

1. 什么是算法？
2. 烹调书中给出的食谱是算法的例子吗？
3. 给出以下每句短语中所描述的计算机部件的名字：
   - 计算机的心脏和最终控制器，所有计算执行的地方
   - 负责在机器内部移动数据的计算机部件
   - 负责所有计算，如加法、减法、乘法及除法的计算机部件
   - 一组线，用于对不同设备进行互连以实现数据连接
   - 用于读、显示或存储数据的设备
   - 用于短期存储数据和执行程序的地方
4. 在一个16位的寄存器中能存储多少种不同的模式？在这样一个寄存器中，能存储为（二进制补码）有符号整数的最大值是什么？最小值是什么？能存储为无符号整数的最大和最小值是什么？
5. 将下面16位二进制数转换成十六进制数和有符号十进制数（你不能使用计算器：）：
   - 1001110011101110
   - 1111111111111111
   - 0000000011111111

- 0100100010000100
- 1111111100000000
- 1100101011111110

6. 将下面32位IEEE浮点数从十六进制转换成标准的十进制表示。
   - 0x40200000
   - 0x41020000
   - 0xC1060000
   - 0xBD800000
   - 0x3EAAAAAB
   - 0x3F000000
   - 0x42FA8000
   - 0x42896666
   - 0x47C35000
   - 0x4B189680
7. 将下面十进制数转换成32位IEEE浮点表示。
   - 2.0
   - 45.0
   - 61.01
   - −18.375
   - −6.68
   - 65536
   - 0.000001
   - 10000000.0
8. 能精确地表示成32位整数但却不能表示成32位IEEE浮点数，存在任何这样的数吗？为什么？
9. 采用标准的ASCII表（查看因特网或附录E），什么样的4个十六进制字节代表串“Fred”？
10. 哪个ASCII字符串与十六进制数0x45617379相对应？
11. 对或错：二进制位数越多，值就越大。为什么？
12. 为什么为Windows奔腾4生成的可执行文件不能运行于基于PowerPC的Macintosh上（在没有特殊的软件支持的情况下）？
13. 虚拟机相比基于芯片的体系结构的最重要的优点是什么？
14. 最重要的缺点是什么？
15. 什么语言可用于为JVM写程序？
16. 对应于下面语句有多少条低级机器指令？

$$x=a+(b*c)+(d*e);$$

1.7 编程习题

以下题用教师同意的任何语言写一个程序。

1. 读入一个32位（二进制）有符号整数，并输出与其等值的十进制数。
2. 读入一个32位（二进制）浮点数，并输出与其等值的十进制数。
3. 读入一个十进制浮点数，并输出与其等值的IEEE 64位十六进制数。（注意：也许需要额外的读入）
4. （以十六进制IEEE格式）读入两个32位浮点数A和B，并以十六进制输出它们的乘积A · B。不要在内部转换成十进制浮点数再利用语言的乘法操作进行乘积计算。
5. （以十六进制IEEE格式）读入两个32位浮点数A和B，并以十六进制输出它们的和A + B。不要简单地转换成十进制再进行加法计算。
6. （以十六进制）读入一个32位浮点数A，并以十六进制和十进制输出其倒数1.0/A。你能使用该程序结合程序4完成浮点除法吗？怎么做？

第2章　算术表达式

2.1　符号表示

2.1.1　指令集

计算机的两个中心问题（也许确实是两个中心问题），是其缺乏想象以及能做的事情有限。考虑用一个典型的袖珍计算器计算下面的问题（图2-1）：

一个底面直径450米，高150米的圆形山的体积是多少？用几分钟查一下几何课本你就会得到公式：圆锥的体积是底面面积和高度的乘积的三分之一。底圆的面积是π乘以半径的平方。半径是直径的一半。π的值当然是3.14多一点。综合起来，答案就是：

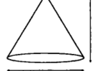


图2-1　一个锥形山

$$\frac{1}{3}\left[\pi\cdot\left(\frac{450}{2}\right)^2\right]\cdot 150$$

你如何计算这个麻烦的公式？

这里就是计算机（在这个例子中就是计算器）的指令集扬眉吐气的地方。这个公式不能按原样输入到一个典型的计算器。它需要被分解成计算器或计算机能处理的很小的片断。只有这种小片断的一个序列才能使我们得到最后的答案。这与传统的计算机编程没有差别，只不过这种计算器所采用的片断比Java等高级语言所用的要小得多。

根据所使用的计算器不同，有若干种不同的按键序列可解决这个问题。在一个典型的高端计算器上，下面的序列可计算$\left(\frac{450}{2}\right)^2$：

[(] [4] [5] [0] [÷] [2] [)] [x^2]

或者，也可用：

[4] [5] [0] [÷] [2] [=] [x^2]

整个计算可如下进行：

[1] [÷] [3] [·] [π] [·] [(] [4] [5] [0] [÷] [2] [)] [x^2] [·] [1] [5] [0]

2.1.2　操作、操作数及顺序

这里隐含了一些微妙的东西。首先注意这个计算器的“指令集”包括一个[x^2]按钮，使得按一下按键就能得到一个数字的平方值。它还有一个π按钮。没有这些便利，就需要更多的按键和更高的复杂性。事实上，几乎没有人知道什么样的按键序列能代替[$\sqrt{x}$]按钮。

其次，这个序列中的顺序很重要。就像450与540之间存在差异一样，450 ÷ 2与450 2 ÷ 之间也存在差异。前一个得出的值是225，而后一个甚至没有意义（在传统的符号意义下）。一般来说，人们想到的多数数学操作是二元的（binary），也就是说，这些数学操作接受两个数字（就是参数，正式的称谓是操作数（operand））并产生第三个数作为结果。3和4相加就得7。

一些高级的数学操作，如sin$x$、cos$x$及log$x$是一元的（unary），意思是它们只接受一个参数。为了处理一个操作，计算机同时需要有定义执行哪种操作的操作符（operator）和所有必需的操作数的值，并且要以非常严格的格式给出。

例如，按传统在黑板上书写数学公式时，二元操作写成中缀（infix）形式，意思是操作符写在两个操作数的中间（如$3+4$）。三角函数如sin写成前缀（prefix）形式，即操作符在操作数之前（如$\sin\frac{\pi}{2}$）。一些高端计算器如惠普HP 49G还支持后缀（postfix）形式（也称为逆波兰表示（reverse Polish notation）），即操作符放在操作数最后。在这样的计算器上，计算450除以2所需要的按钮序列是：

[4][5][0][ENTER][2][÷]

虽然这个表示乍看起来有些令人费解，但很多人在有了一点经验后更乐于使用它，其原因我们将在后面介绍。

2.1.3 基于堆栈的计算器

这种类型的计算器（采用逆波兰表示）采用称为堆栈（stack）的数据结构很易于建模。这个术语来源于自助餐厅中叠在一起的盘子或快餐馆中的泡沫塑料杯（见图2-2）所呈现的景象。这样的杯子通常摆在弹簧支撑的容器中，杯子的重量将整个一堆杯子压下使得顶上的杯子总是保持在一个不变的高度。在任何时候，只有顶上的杯子被移走（导致堆的其余部分向上略微弹出并露出一个新的杯子）或者可在顶上再放进一个杯子将杯子略微向下压。用稍微抽象的术语说，只有顶上的对象是可访问的。

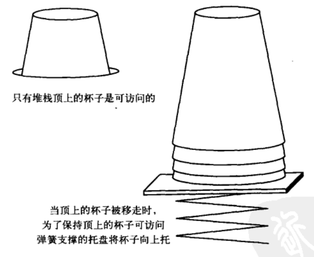


图2-2 自助餐厅的一堆杯子

堆栈是一组具有类似访问性质的数字或数据对象。只有“顶部”的对象是可处理的，它可以在任何时候被移走（“弹出”），另外的一个对象可以加入（“压入”）堆栈而取代原来的顶部对象成为新的顶部对象（原来的顶部对象就处于距顶部的第二个位置，依次类推）。基于堆栈的计算尤其适用于后缀表达式。各个项依次压入堆栈。对于像sin和cos这样的一元操作，可简单地弹出堆栈顶部的项作为操作数，然后执行这个操作，再将结果压入。对于像加法这样的二元操作，就弹出堆栈顶部的两项，然后执行操作，再将结果压回堆栈。

下面的操作序列执行了上面所描述的计算：

1 3÷ π · 450 2 ÷450 2 ÷ · 150.

打开这个序列，前面的数字1和3被压入堆栈。然后，这两个数被弹出并计算出其商为1/3即0.3333。压入π值，然后0.3333和π相乘，得1.047，依次类推。计算可按序列快速执行，无需使用括号和结构。特别要注意，操作的顺序不会有二义性，而这种情况却可能在中缀表达式中出现，比如1＋3 · 4。两种可能的等价表达式是1 3 4 · ＋和1 3 ＋ 4 · 显而易见是不同的。

2.2 存储程序计算机

2.2.1 取指－执行周期

计算机在进行计算时，当然不需要人通过操作设备（通过按钮）直接与其打交道，而是将每个可能的操作存储成位模式（如前面章节所述）。操作的序列就存储成包含位模式序列的程序。计算机通过读存储器（到控制单元）而获得指令，并将每个位模式解释成要执行的操作。这通常称为取指-执行周期（fetch-execute cycle)，因为这种执行是周期性的和无限的，在计算机中每秒要运行几百万或几十亿次（见图2-3）。

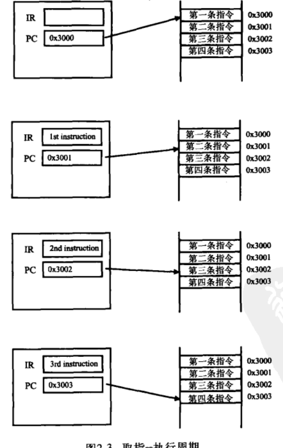


图2-3　取指–执行周期

控制单元内部至少有两个重要的存储位置。第一个是指令寄存器（instruction register）即IR，它保存了刚从存储器中取来的位模式，因而该位模式可以得到解释。第二个是程序计数器（program counter）即PC，它保存了要取出下一条指令的地址。在正常情况下，每次取出一条指令，PC递增并指向存储器中下一个字。例如，如果PC包含模式0x3000，则下一条指令就将从这个地址取出（解释成一个存储器地址）。这意味着存储在地址0x3000处的位模式（不是0x3000本身）将被放到IR。PC然后得到更新，存储的新值是0x3001。在以后的连续周期中，PC将包含0x3002、0x3003、0x3004等等。在每个这样的周期中，IR包含计算机要执行的指令。指令的典型例子包括数据传输（数据在CPU和存储器或I/O外设间移动）、数据处理（对已经在CPU中的数据做一些算术和逻辑操作），或者影响到控制单元本身的控制操作。

对指令加以解释本身是一个中等复杂度的任务，部分原因是某些指令实际上是需要进一步详细说明的指令组。例如，将信息存到存储器的指令是不完备的。什么信息要存储？要存储到存储器中的哪个位置？此外，很多机器需要更多的细节，如“寻址模式”（这个位模式指的是一个存储器位置还是指一个数呢？）、要存储的数据的大小（字数）等等。基于这个原因，很多机器语言指令实际上是相关位的综合体（就像前面一章中浮点数的详细结构一样）。

针对IBM-PC（Intel 8086及更晚的机器）的一个例子是简单的ADD指令。这个指令可编码为两个字节，其中第一个8位保存的数是0x04，第二个8位保存一个无符号的从0到255的8位数。这个数隐含地将被加到已经存储在特定位置（即AL寄存器，CPU中一个特殊的8位寄存器）的数上。如果你需要加一个大于255的数，就要用一个不同的机器指令，这个指令的第一个字节是0x05，接下来的16位定义了一个从0到65 525之间的数。如果你想将数加到一个不在AL寄存器的数中，则机器指令就以操作代码字节0x81开头，第二个字节定义数存储的地址，然后定义将要被加的数。

由于JVM体系结构所选择的设计（特别是，所有加法都在一个地方（即堆栈中）完成），使得在JVM上的相应解释任务较为容易（见图2-4）。下一节中将会讨论到，这既反映了设计者的基本设计理念也反映了JVM本身的能力。例如，所有的加法都在堆栈上完成（就像在RPN计算器中一样）。这意味着计算机不需要担心加数来自哪里（因为它们总是来自堆栈）或者相加所得的和送到哪里（因为它总是送回到堆栈）。这样，两个数相加的指令就是一个单字节的值0x60，不会混淆。

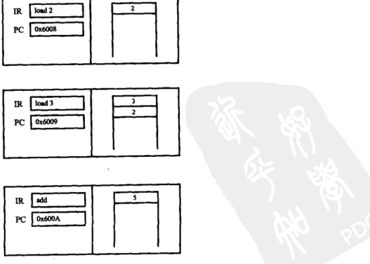


图2-4 在JVM上基于堆栈的计算示意图

补充资料

存储程序计算机和冯·诺依曼体系结构

第一台计算机产生于第二次世界大战中的弹道计算器和密码破译机。将它们称为计算机几乎是不对的，因为它们实际上仅仅是复杂的专用电气设备。最重要的是，要改变它们的用途，就需要改变机器的物理电路。程序是“硬连线”到计算机中的。如果需要求解一个不同的问题，你就要重连机器并建立新的电路。

伟大的数学家约翰·冯·诺依曼认识到了这个重大的局限性并为建造“通用”计算机提出了（在1946年与Arthur Burks和Hermann Goldstine合写了“电子计算设备逻辑设计的初步讨论”）一个革命性的概念。他确定了计算中涉及的4个主要的“器官”，分别与运算、存储、控制以及与外部世界（和人类操作者）连接有关。根据他的观点，建立通用计算机的关键是：计算机应该不仅能存储算术运算的中间结果，还能存储产生这些计算的（指令）顺序。换句话说，“控制器官”应该能从存储器中读入模式并依次行动。而且，控制模式的存储应该与数字数据的存储一样灵活。因此，为什么不像存储数字数据一样将指令也存储成二进制模式呢？这样，控制器官的设计就变成一种选择器：当该模式被从存储器中读出时，就激发相应的电路。冯·诺依曼进一步指出，如果存在某种方式能加以分辨，控制模式和数据模式甚至能处于同一存储器中。与此不同的另一种方式（现代计算机所采取的途径）则是，两者的区分不是通过模式，而是通过用途。任何装入到控制单元中的模式就自动被认为是指令。（一个与其竞争的体系结构称为哈佛体系结构（Harvard architecture），它采用分开的存储器分别存储代码和数据。我们将会在后面讨论Atmel微控制器时看到这种体系结构）。这也意味着指令可作为数据使用，甚至能被覆盖，使得自修改代码（self-modifying code）成为可能。这使得计算机可自行重编程，比如，通过将程序（作为数据）从外存储器拷贝到主存储器，然后（作为代码）执行程序。

冯·诺依曼计算机的操作是通过重复地执行以下操作完成的：

1) 从存储器官中取得一个指令模式。
2) 确定并从存储器中取得该指令所需的数据。
3) 在运算器官中对数据进行处理。
4) 将运算的结果存储到存储器官中。
5) 返回到步骤1。

冯·诺依曼就这样奠定了当今计算机大部分的理论基础。例如，他提出的四个器官可以容易地与ALU、系统存储器、控制单元以及较早定义的外设对应起来。他提出的操作方法就是取指－执行循环。研究者们这几十年一直在探究冯·诺依曼体系结构，并且已能在某些方面突破他的模型的限制。例如，一般来说，多处理器系统用几个CPU代替单控制器官，每个CPU可独立操作。一个更激进的非冯·诺依曼体系结构可在各种神经网络和连接系统设计中看到，在这种系统中，“存储器”分布在几十到几千个互锁的“单元”中，并且没有控制器官。今天，“冯·诺依曼计算机”这个术语已经很少提及了，就像鱼不会常常谈论水一样。这种计算机无所不在，所以通常假定任何给定的机器都遵循冯·诺依曼/存储程序体系结构。

2.2.2 CISC计算机与RISC计算机

显然，CPU要做的不同事情越多，操作代码和机器指令就越多。不同的操作代码越多，

需要的位模式就越多，就需要（平均）更长的位模式以确保计算机能将它们分辨开来。（想象一下，如果0x05不仅是“将两个数相加”的意思，还是“停机”的意思，会发生什么样的灾难！计算机怎么能区分出它究竟是什么意思呢？）然而，更长的操作代码就意味着有更大的IR，就意味着有更大的总线将计算机连接到程序存储器，也就意味着有更复杂和昂贵的电路来解释指令。因为每个操作可能需要不同的连线和晶体管，这样，一个复杂的指令集就需要有很多的电路来执行各种操作。这意味着这种复杂的CPU芯片会很昂贵，需要高成本的支持，并且与更简化的设计相比每条指令的运行更慢。（而且还需要更大的芯片，运行时更热[因此需要有更好的冷却系统]，消耗更大的功率，降低了电池的寿命。）

这就意味着更小的指令集就更好吗？不一定，因为虽然具有精简指令集的CPU可能对某些任务运行得更快（例如，每个CPU都需要完成两个数相加的能力），但还有很多任务，较小的CPU需要几个步骤才能完成。例如，一个复杂指令集计算（complex instruction set computing, CISC）芯片可能会将一块数据，比如包含有几千字节的串，从存储器中的一个位置移动到另一个位置而无需使用CPU内部的任何存储器。与此相比，一个精简指令集计算（reduced instruction set computing, RISC）芯片可能就不得不将每个字节或者字先从存储器移动到CPU，然后再移动到存储器。更重要的是，在每个步骤，移动字节的指令都要被取出和解释。所以，虽然总体上取指－执行周期可以运行得更快（通常也确实是这样），但特定的程序或应用却需要用很多指令（很慢地）完成任务，而CISC计算机却可以用单条（虽然长且复杂）的指令完成同样的任务。RISC支持者所声称的另一个优势是对“滋长特性主义”（creeping featurism）（即设备和程序为了有新特性而增加复杂度）的抵御。一个RISC芯片通常有一个小的、纯净的并且定义简单的设计，并且在未来的版本中将保持这种小而纯净且简单的特点。而CISC的较新版本通常会加入更多的指令、更多的特性以及更高的复杂性。这会带来很多影响，其中之一就是阻碍了向后兼容性，因为给较晚CISC写的程序通常会使用那些六个月之前的CISC中不存在的特性和指令。当然，在另一方面，新加入的特性也许对工程师们是有用的。

当前市场上的两款主要的CPU芯片为这种差异提供了很好的例证。奔腾4是CISC芯片，具有巨大的指令集（甚至不考虑使用哪个寄存器或存储器位置保存数据，单是ADD就已有34种不同的表示方式），而PowerPC是一个RISC芯片，其设计是为了能快速执行常见的操作，对罕见或复杂任务则用时要长得多。因为这个原因，在比较两个CPU间的处理速度时，我们不能仅仅去看时钟速度这样的数字。在RISC芯片上的单个时钟周期一般对应于单个机器指令，而在CISC上做任何事一般都要至少两个或三个时钟周期。另一方面，根据应用的不同，CISC中较大的指令能在较少的时钟周期内做更复杂的计算。这样，性能的差异就更取决于正在运行的程序类型和具体的操作而不是时钟速度的差异。特别地，苹果公司的销售文件声称，仅仅通过在每个时钟周期做更多事情，（RISC）865MHz的Power Mac G4就比（CISC）1.7GHz的奔腾4运行起来平均要快大约60%（图形和音频操作要快3到4倍...）。不管你是否相信苹果公司的销售文件，他们的中心点（即时钟速度并不适合比较不同的CPU，另外计算机的指令集可按不同种类的任务进行设计）仍几乎是不可辩驳的，而无论在这场CISC/RISC争论中你站在哪一边。

2.3 JVM上的算术运算

2.3.1 一般评述

Java虚拟机是基于堆栈的RISC处理器的一个示例，但要注意其在物理上是不存在的。像

多数计算机一样，因为效率的原因，JVM的直接算术运算能力限于常见和简单的操作。很少有计算机提供三角函数，JVM也是这样，但所有计算机都支持基本的算术运算，如加法、减法、乘法及除法。然而，通过采用基于堆栈的计算（就像较早讨论过的高端计算器），JVM超出了多数的计算机。这使得在其他计算机上进行仿真非常容易，而固定数量的寄存器是不能实现这种仿真的。JVM本身维护着一个保存二进制模式的堆栈，维护着先前压入的元素和早先计算的结果。任何操作的操作数都取自于堆栈顶部的元素，计算结果也返回到堆栈的顶部。要计算7·（2+3），则将7、2、3（按顺序）压入堆栈，然后，先执行加法再执行乘法。

因为JVM（以及一般意义上的Java）采用有类型（typed）的计算，所以这个过程实现得更复杂一些（在这一点上，它与多数的RISC机有点不同，但它提供的安全性高得多）。就像在前一章所讨论过的，相同的位模式可表示若干个不同的项，而相同的数可用若干种不同的位模式表示。为了准确地处理这些问题，任何计算机都需要知道由位模式表示的数据类型。JVM仅仅是在如何严格地保证这种类型系统以防止程序错误方面显得不同一般。

因此，根据加法操作数的类型的不同，就有若干种不同的方式来表示加法。（这也许就将JVM置于CISC/RSIC争论的中间某个地方）。一般来说，操作名的首字母就反映了所期望的参数类型和结果类型。要将两个整数相加，就使用助记符`iadd`，而要将两个浮点数相加，就使用助记符`fadd`。其他算术操作也遵循这个模式，所以整数相减的操作就是`isub`，而两个双精度数相除的操作就是`ddiv`。

详细地看，JVM堆栈是一组32位数的集合，没有固定的最大深度。对于可以放入32位寄存器的类型，将数据元素存到一个堆栈位置没有问题。更长的类型通常存储为成对的32位数，所以一个64位的“双精度”数在堆栈顶部实际上占用两个位置。

JVM堆栈上有多少个位置？理论上，因为JVM没有硬件限制，你需要多少就有多少。实际上，你写的每个程序、方法或函数都会定义一个最大堆栈大小。类似地，对所需的存储器大小没有硬件定义的限制。与此不同，每个方法都定义了局部变量的最大数量，这些变量不是存储在堆栈上，而是用于临时存储值。

2.3.2 一个算术指令集示例

数据类型

JVM支持8种基本数据类型，其中多数与JAVA语言本身的基本数据类型密切对应。这些类型列在表2-1中。JVM还提供了大多数的标准算术操作，包括学生们可能不熟悉的一些操作。为了精简指令集，做了某些设计简化。

表2-1　JVM的基本类型及其表示

类型	代码	解释
int	i	32位带符号整数
float	f	32位IEEE 754浮点数
long	l	64位整数，存储在两个连续的堆栈位置
double	d	64位IEEE 754浮点数（如上）
byte	b	8位带符号整数
short	s	16位带符号整数
char	c	16位无符号整数或UTF-16字符
address	a	Java对象

值得注意的是这一组类型中没有布尔（boolean）类型，布尔类型在Java中但不在JVM中。

布尔变量当然只能存有真或假值，并可用单个位来实现。然而，在大多数计算机中，访问单个位要比访问一个字低效得多。为此，在JVM中，布尔值简单地表示成字大小（32位）的0或1，也就是整数。

小于字的存储类型，如字节、短整数及字符，相当于次级类型。由于在JVM中做32位数学计算不会比做更小数的数学计算用的时间更长，故所有这种数据类型的变量就自动在CPU内提升为32位整数。另一方面，这种类型的变量在存储时有一个明显的不同。例如，一个1百万字节的数组所占用的空间是类似整数数组的四分之一。由于这个原因，JVM支持小类型（字节、短整数、字符甚至布尔）从存储器中读出或存到存储器，特别是对于数组的存取。

基本算术操作

对于这些需要通过特殊的处理来支持的数据类型，几乎每个类型和操作的组合都需要一个特定的操作代码和助记符。为简化程序员的任务，多数的助记符都使用字母码来指明动作的类型。例如，将两个整数相加的助记符是`iadd`，将两个长整数相加就要用`ladd`，而要将两个浮点数和双精度数相加就分别用`fadd`和`dadd`。为简便，这个系列简写成`?add`，其中`?`代表任何合法的类型字母。

基本的算术操作加（`?add`）、减（`?sub`）、乘（`?mul`）、除（`?div`），对4种主要的类型（整数、长整数、浮点数及双精度数）都进行了定义。所有这些操作都是通过从堆栈位置弹出前两个“元素”（注意，对于长整数或双精度数的操作，前两个“元素”的每个元素都涉及两个堆栈位置，因此总共就要四个堆栈位置），计算结果，然后将结果压回到堆栈顶部。此外，JVM提供`?neg`操作，其功能是对堆栈顶部项的符号取反。这当然也可以通过将值−1压入堆栈然后执行一个乘法指令来实现，但对于这个常见的动作，用一个专用的操作会执行得更快。

`?div`操作有一个方面需要引起特殊注意。`idiv`和`ldiv`都需要对整数操作并产生整数作为结果（没有分数或小数）。例如，8除以5产生结果1，而不是浮点数1.6。要执行一个浮点除法，就有必要先将两个参数转换成浮点或双精度类型，这在后面将要讨论。类似地，对整数和长整数类型有一个特殊的操作`?rem`，其功能是取余数或模（modulus）。这个操作对于浮点/双精度类型不存在，因为除法操作要执行的是精确的除，就是说要精确到机器表示所允许的程度。

逻辑操作

JVM只为整数和长整数类型提供了基本的逻辑操作：与（`?and`）、或（`?or`）及异或（`?xor`）（见图2-5和图2-6）。这些操作以按位（bitwise）的形式进行，意思是第一个操作数的每个位都独立地与第二个操作数的相应位进行操作，结果就是各个独立位操作结果的总和。当应用到布尔值时，0与/或1，结果正如所料。0的表示是0x0000，1的表示是0x0001。对于除了最后一位的位置，相应的位是0和0；对于最后一个位置，相应的位就是0和1。值0x0000与0x0001进行OR操作就等于0x0001。换句话说，假OR真是真，正如预期。

第一个值	1	1	0	0	1	0	1	0	= 0xCA
第二个值	1	1	1	1	0	0	0	0	= 0xF0
AND结果	1	1	0	0	0	0	0	0	= 0xC0

图2-5 按位AND操作示意图

第一个值	1	1	0	0	1	0	1	0	= 0xCA
第二个值	1	1	1	1	0	0	0	0	= 0xF0
XOR结果	0	0	1	1	1	0	1	0	= 0x3F

图2-6 按位XOR操作示意图

移位操作

除了这些熟悉的操作外，JVM还为改变位模式尤其是在数中移动位而提供了一些标准的移位（shift）操作。在Java/C/C++中，这些都表示成<<或者>>操作符。基本的操作就是将数中的每一位向右或向左移动一个位置。以二进制模式0xBEEF为例：

B	E	E	F	
1011	1110	1110	1111	becomes
0111	1101	1101	1110	左移一位
0101	1111	0111	0111	右移一位

这样，0xBEEF左移一位就成为0x7AAE，右移一位就成为0x5F77。在两种情况下，右边或左边留下的空位都用0来填充。这种移位称为逻辑移位（logical shift），与算术移位（arithmetic shift）相对应。算术移位试图保留数的符号，所以在算术右移时，要不断地复制符号位填充在空位上。(算术左移无需复制最右边的位)。

B	E	E	F	
1011	1110	1110	1111	becomes
0101	1111	0111	0111	逻辑右移
1101	1111	0111	0111	算术右移

特别对于有符号数的情况，逻辑右移的结果总是正数，因为0被插入到了最左边的符号位。相比之下，当且仅当原始值是负数时，算术右移的结果总是负数，因为在操作时符号位被复制。对于无符号数，左移等价于将该数乘以2的某次幂，而逻辑右移等价于将该数除以2的某次幂。一般来说，这些操作用于将一组已知的位放置于模式中的特定位置上，例如用来作为将要进行的按位AND、OR、XOR操作的一个操作数。JVM为执行这些移位提供三种操作：?shl（左移）、?shr（算术右移）、及?ushr（逻辑右移，这个助记符实际上表示“无符号右移”），这些操作对整数（32位模式）和长整数（64位模式）都适用。

转换操作

除了这些基本的算术和逻辑操作，还有一些形式如?2?的一元转换操作。例如，i2f就是将一个整数（i）转换成一个浮点数（f）。一般来说，每个这种操作都是弹出堆栈顶端的元素，将其转换成合适的新类型，并将结果压入堆栈。这通常不会改变堆栈的总体大小，除非转换是从长类型到短类型（或相反）。例如，操作i2l将从堆栈弹出一个字（32位），将其转换成64位（2个字），然后将两个字的数压入堆栈，占两个元素。所产生的效果就是使堆栈的深度增加1，类似地，d2i操作会将堆栈深度减小1。

如前所述，因为效率的原因，不是所有的类型组合都为JVM所支持。一般来说，转换到整数或从整数转换到其他类型的数总是可以的。在四种基本类型整数、长整数、浮点数、双精度数之间进行转换也总是可以的。然而，从一个字符就不能通过一次操作直接转换到一个浮点数，反过来也不行。这有两个主要原因。首先，由于小于字（sub-word）的类型自动地转换成字类型，所以这个本来要定义成b2i的操作在某种意义上说是自动进行，甚至是不可避免的。其次，如前所述，将整数转换成浮点数（例如，2↔2.0）不仅涉及从更大的整体中选择特定的位，还采用了完全不同的表示系统，并改变了表示的基本模式。如果需要在一个浮点数和一个字符之间进行转换，就可通过两个步骤来完成（f2i，i2c）。由于类似的原因，三种操作i2s、i2c、i2b的输出有些不寻常。操作结果不是生成（并压入）助记符中的第二个类型的数，而是生成一个整数。然而，生成的整数将被截断成适当的大小和范围。这样，在0x24581357上执行i2b操作就会产生模式0x00000057，等价于单字节0x57。

2.3.3 堆栈操作

无类型堆栈操作

除了这些特定类型的操作，JVM还为日常堆栈处理提供了一些通用和无类型的操作。一个简单和明显的例子就是pop操作，其功能是从堆栈弹出单个字。由于这个值就要被丢弃，所以它是一个整数、一个浮点数、一个字节或者其他类型都不要紧。类似地，pop2要从堆栈移除两个字，或者就是一个双字的条目、或者是一个长整数、或者是一个双精度数。这种类型的类似操作还包括dup，其功能是在堆栈顶端复制并压入单个字的条目。dup2则是复制和压入一个双字的条目。swap的功能是交换堆栈顶端的两个字。nop是“无操作”的缩写，它不做任何事情（但要占用时间，所以在需要使机器等待一微秒左右时间时可用这条指令）。

此外，还有一些不常用的操作，用来执行相当特殊的堆栈操作。比如dup_x1，其功能是复制堆栈顶端的字然后将其插入到第二个字的下面。如果堆栈自顶向下保存的值是（5 3 4 6），则执行dup_x1就会产生（5 3 5 4 6）。这些特殊的操作见附录B，在本书中就不做进一步讨论了。

常数和堆栈

当然，为进行这些基于堆栈的计算，就需要用一些方法首先将数据置于（压入）到堆栈上。根据要压入的数据类型以及该数据来自于哪里，JVM有若干个这种方法。

最简单的情况是将一个常数压入到堆栈。根据该常数的大小不同，你可以使用bipush指令（压入一个字节的带符号整数）、sipush指令（2字节带符号整数）、ldc指令（一个单字的常数，如一个整数、一个浮点数或一个地址）、或者ldc2_w指令（一个双字的常数，如一个长整数或一个双精度数）。所以，将整数3和5压入到堆栈，然后将它们相乘的代码就如下所示：

```
bipush  5
bipush  3
imul
```

其变型：

```
sipush  5    |    ldc 5
sipush  3    |    ldc 3
imul         |    imul
```

能完成同样的事情，但效率比较低。因为5和3很小，可以装入到单个字节中并用bipush指令压入。而且还要注意，因为乘法是可交换因数位置的，所以你先压入5还是先压入3没有关系。减法和除法就不是这种情况，在这两种操作中，计算机从先压入的数中减去后压入的数，或者将先压入的数除以后压入的数。所以，在上面例子中，用idiv取代imul将导致5除以3，在堆栈上产生结果1（不是1.66667，因为idiv规定的是整数除，小数部分被舍掉）。

为了提高效率，有若干种专用操作能将常用的常数以更快的速度压入到堆栈。例如，iconst_N，其中N是1、2、3、4、5之一，就能将一个字的整数压入到堆栈上。由于常常有必要将一个变量初始化成1或者将一个变量加1，iconst_1要比实现同样功能的bipush 1要快。这样，我们就可将上面例子重写成如下代码使其运行得稍微地快一些：

```
iconst_5
iconst_3
imul
```

类似地，iconst_m1将整数值-1压入。等价的快捷表示还有针对浮点数的（fconst_N对于0、1、2）、长整数的（lconst_N对于0、1）、以及双精度数的（dconst_N对于0、1）。

局部变量

除了装入常数，还可以从存储器装入值。每个JVM方法都有一组可自由随机并以任何顺序

访问的存储器位置与其关联。像堆栈一样，可得到的存储器位置不受硬件的限制，可由程序员设定。而且，如前所述，装入的模式类型决定了操作及助记符。要装入一个整数，就用iload；要装入一个浮点数，就用fload。各操作都能够取得适合的变量并将其值压入到堆栈顶部。

变量利用顺序的数字来引用，从0开始。所以，如果一个给定的方法有25个变量，就有0到24的数字。每个变量存储了一个标准的字大小的模式，所以变量没有类型。双字模式（长字或双精度字）的存储就稍微更复杂一些，因为每个模式必须占用两个相邻的变量。例如，从变量4装入一个双精度数，实际上是从两个变量4和5读取值。此外，JVM还允许若干种像?load_N这样的快捷方式，所以用iload 0或者快捷方式?load_0都能从局部变量0（#0）装入一个整数值。对于所有4种基本类型、从0到3的所有变量都存在这个快捷方式。

类似地，数据可从堆栈弹出，并存储在局部变量中以备后用。这种情况采用的指令是?store，其中首字符仍是存储类型。如前所述，存储一个长整数或双精度数实际上要执行两次弹出操作，并将结果存储在两个相邻的变量中。所以，指令dstore 3要从堆栈中移出两个元素而不是一个，并改变两个局部变量#3和#4。同样，如前所述，对所有类型并且N从0到3变化时都存在形如?store_N的快捷方式。

2.3.4　汇编语言和机器码

让我们看一个简单的代码片断来了解各种转换如何发生。我们将从单条简单的高级语言语句开始（如果“简单高级语言语句”在术语上没有矛盾）：

```
x = 1 + 2 + 3 + 4 + 5;
```

这条语句（很清楚吧？）要计算常数1到5的和并存入局部变量x。首先的问题是JVM不理解命名局部变量的思想，只理解编号的变量。编译器需要识别出x是一个变量，并为其分配相应的编号（假设使用#1并且它是一个整数）。执行这个任务所需要编写的一种代码（还有很多其他写法）就是如图2-7所示的操作序列。

```
; x = 1 + 2 + 3 + 4 + 5;
; 转换成：1 2 + 3 + 4 + 5 +，然后装入到#1
iconst_1          ; 装入常数值1
iconst_2          ; 装入常数值2
iadd              ; 做加法
iconst_3          ; 装入常数值3
iadd              ; 做加法
iconst_4          ; 装入常数值4
iadd              ; 做加法
iconst_5          ; 装入常数值5
iadd              ; 做加法
istore_1          ; 存到x
```

图2-7　jsmin程序片断#1

这是编译器的主要任务，就是将高级语句转换为若干个基本操作。到此，汇编器（或者编译器的一个不同部分）的任务就是将每个操作助记符转换为相应的机器码字节。全部的对应关系在附录B和C中给出。我们注意到该程序用到的iadd对应于机器指令0x60。转换全部程序就产生表2-2所列出的结果。

这样，相应的机器码就是字节序列0x04、0x05、0x60、0x06、0x60、0x07、0x60、0x08、0x60、0x3c，这些将被存储到磁盘上作为可执行程序的一部分。

转换并不总是这么简单，因为一些操作要占用多个字节。例如，bipush指令将一个字节压入到堆栈上（就像iconst_0将0压入到堆栈上一样）。但是，是哪个字节呢？bipush指令本

身具有值0x10，但它后面总是跟一个字节，表明需要压入的是什么。要压入值0，编译器也可用bipush 0，这将被汇编成两个连续的字节：0x10、0x00。这样，我们就有了该程序的另一种版本，如表2-3所示。注意，这第二个版本大约要加长50%，因此更低效。

表2-2　将程序片断#1转换成字节码

助记符	机器码字节	助记符	机器码字节
iconst_1	0x04	iconst_4	0x07
iconst_2	0x05	iadd	0x60
iadd	0x60	iconst_5	0x08
iconst_3	0x06	iadd	0x60
iadd	0x60	iconst_1	0x3c

表2-3　将程序片断#2转换成字节码

修改的助记符	机器码	修改的助记符	机器码
bipush1	0x10	bipush4	0x10
	0x01		0x04
bipush2	0x10	iadd	0x60
	0x02	bipush5	0x10
iadd	0x60		0x05
bipush3	0x10	iadd	0x60
	0x03	istore_1	0x3c
iadd	0x60		

2.3.5 非法操作

因为堆栈和局部变量都只存储位模式，程序员可能很容易弄混。这尤其可能发生在压入常数时，因为常数的类型不是明确地标记在助记符中。将整数值10置于堆栈的命令是ldc 10，将浮点值10.0置于堆栈的命令是ldc 10.0。由于大多数JVM汇编器足够聪明，能够区分出10是一个整数而10.0是一个浮点数，所以在两种情况下都能将正确的位模式压入到堆栈。然而，这些位模式是不一样的。试图压入两次10.0然后执行一个imul指令将不会得到100.0（或者甚至100）。试图将浮点数当成整数执行一个算术运算，或者将整数当成浮点数执行一个算术运算都是错误的。在最好的情况下，机器会给出警告。在最坏的情况下，机器甚至不理会，并给出完全错误的答案，而且这种错误是神秘而不可追踪的。

同样，企图将双字变量、长整数或双精度数当成单字变量来访问其上半部分（或下半部分）也是错误的。如果你已经将一个双精度数存储到局部变量#4（及#5），就不能从局部变量#5（#4）装入一个整数。最好情况下机器仍然会给出警告，最坏情况是默默地给出无意义而且极其错误的结果。试图从空堆栈弹出、从不存在的变量装入或存入不存在的变量等等也是错误的。

JVM的主要优势之一就是其设计（第三章将会讨论）能够在这类错误发生时，甚至在写程序时就捕获它们，使得程序员得以摆脱错误。这就极大地提高了JVM程序的安全性和可靠性。然而，一个好的程序员不应该依赖计算机的能力来捕捉错误。精心地规划和谨慎地编写代码是确保计算机得出正确结果的更好方式。

2.4 一个样例程序

2.4.1 一个有注解的例子

回到本章开头的圆锥体问题，现在问题就变成我们不仅想知道问题的答案，而且还要知道如何在所讨论的机器（JVM）上实现。简要回顾一下，原问题是：

一个底面直径450米，高150米的锥形山的体积是多少（见图2-8）？这个公式在黑板上可以写作：

$$\frac{1}{3}\cdot\left[\pi\cdot\left(\frac{450}{2}\right)^2\right]\cdot 150$$

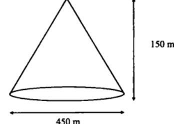
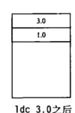


图2-8　一个锥形山

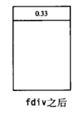
用什么JVM指令序列能解决这个问题呢？

首先，我们需要计算浮点值1/3。整数除不行，因为1/3等于0，而1.0/3等于0.333333。这样，我们将两个元素1.0和3.0压入并执行一个除法：

```
ldc 1.0
ldc 3.0
fdiv
```



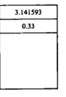
ldc 3.0之后



fdiv之后

然后压入已知的π值。

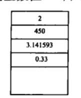
```
ldc 3.141593
```



ldc 3.141593之后

要计算半径，我们只要压入450，压入2，然后再做除法。注意450过大，不能存储在单个字节中（单字节最大也只能到整数值127），所以必须用sipush。

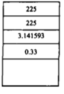
```
sipush 450
bipush 2
idiv
```



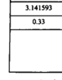
bipush 2之后

要对半径求平方，我们可以重新计算 $\frac{450}{2}$ ，或者也可更高效地采用dup指令拷贝堆栈顶部元素并执行乘法。

```
dup
imul
```



dup之后



imul之后

下面，将高度150再压入并做乘法。

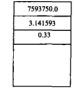
```
sipush 150
imul
```

sipush 150之后
150
50625
3.141593
0.33

imul之后
7593750
3.141593
0.33

堆栈顶部的整数值必须转换成浮点数，然后两个（浮点）乘法就计算出最终答案并将其置于堆栈顶部。

```
i2f
fmul
fmul
```



i2f之后

这个过程将被存储成一个机器码指令序列。每个指令将分别从存储器中取出并执行。由于语句是顺序的，下一条要取出的指令就是指令序列的下一条指令，这样，这个复杂的语句序列就会按期望的那样工作。在指令集可利用的基本操作范围内，可以用类似的过程完成任何计算。

2.4.2 最终的JVM代码

```
; 计算1/3
ldc    1.0
ldc    3.0
fdiv

; 压入pi
ldc    3.141593

; 计算半径
sipush 450
bipush 2
idiv

; 对其求平方
dup
imul

; 压入高度
sipush 150

; 将高度与半径的平方相乘
imul

; 转换成浮点数
i2f
; 再乘以以前计算出的π和1/3
fmul
fmul
```



2.5 JVM计算指令总结

算术操作：

	整数	长整数	浮点数	双精度数	短整数	字符	字节
加法	iadd	ladd	fadd	dadd			
减法	isub	lsub	fsub	dsub			
乘法	imul	lmul	fmul	dmul			
除法	idiv	ldiv	fdiv	ddiv			
求余数	irem	lrem					
取反	ineg	lneg	fneg	dneg			

逻辑（布尔）操作：

	整数	长整数
逻辑AND	iand	land
逻辑OR	ior	lor
逻辑XOR	ixor	lxor

移位操作：

左移（算术）

整数	长整数
ishl	lshl

右移（算术）

整数	长整数
ishr	lshr

右移（逻辑）

整数	长整数
iushr	lushr

转换操作：

从： 到：	整数	长整数	浮点数	双精度数	短整数	字符	字节
整数		i2l	i2f	i2d	i2s	i2c	i2b
长整数	l2i		l2f	l2d			
浮点数	f2i	f2l		f2d			
双精度数	d2i	d2l	d2f				

（将短整数、字符及字节转换到整数是隐含和自动的）

2.6 本章回顾

- 计算机与计算器一样，只能做其硬件所允许的一些动作。对于不能用单个操作或按钮点击完成的复杂计算，就必须用若干个可允许的基本步骤组成的序列来完成。
- 传统的数学，如写在黑板上的数学计算公式，采用的是中缀表示，其中像除法这样的操作符写在两个参数之间。而一些计算器或计算机（JVM就是其中之一），采用的是后缀表示，操作符放在参数之后。
- 后缀表示可容易地用称为堆栈的数据结构描述和模拟。
- 任何计算机的基本操作都是取指－执行循环，在此过程中，指令被从主存储器中取出、解释并执行。这个循环的重复次数没有限制。
- CPU拥有为进行取指－执行周期所需要的两个重要信息：指令寄存器（IR）保存的是当前正在执行的指令，程序计数器（PC）保存的是下一条要取出指令的地址。
- 计算机有两个基本的设计思想：复杂指令集计算（CISC）和精简指令集计算（RISC）。Intel 奔腾和苹果/IBM/Motorola Power体系结构分别是体现这两种思想的典型实例。
- JVM采用有类型基于堆栈的计算来执行大多数的算术操作。助记符描述了要执行的操作以及所使用的数据类型。在错误类型的数据上执行一个操作将会产生错误。

- JVM还对经常使用的操作提供了一些快捷操作，比如将值0装入到堆栈。
- 简单的基本数学操作序列能执行非常复杂的计算。

2.7 习题

1. 在计算器上有一个$\boxed{\sqrt{x}}$按钮有什么优点？有什么缺点？
2. 在一个标准的（中缀表示）计算器上，用什么样的操作序列来计算$(7+1)\cdot(8-3)$？在一个RPN后缀计算器上怎么做呢？
3. 是否存在相应的采用前缀表示的操作序列来执行上述计算？如果有，是什么？如果没有，为什么？
4. 取指－执行周期在CISC或RISC计算机上哪一个运行更复杂一些？为什么？
5. 有类型计算与无类型计算之间的差别是什么？分别给出两个例子。
6. 有类型算术运算的优点和缺点各是什么？
7. 为什么不存在cadd指令？
8. 下面的指令哪些是非法的？为什么？
   - bipush 7
   - bipush -7
   - sipush 7
   - ldc -7
   - ldc2 -7
   - bipush 200
   - ldc 3.5
   - sipush -300
   - sipush -300.0
   - ldc 42.15
9. 怎样用移位操作将一个整数乘以8？
10. 描述两种方式将一个64位长的整数中的最低8位提取出来。
11. 存在能在整数和浮点数上都能运行的操作吗？整数和长整数呢？
12. 在?div指令中，被除数是存储在堆栈顶端还是从顶端开始的第二个元素？
13. 球表面的面积是相同半径的圆面积的4倍。写出一个后缀表达式来计算半径为R的半球圆顶的表面面积。
14. 写出一个后缀表达式来计算a、b、c、d及e五个数的算术平均值。
15. 证明对任意的中缀表达式都存在等价的后缀表达式。反之亦然。

2.8 编程习题

1. 写一个程序，实现功能：解释一个后缀表达式并输出结果值。
2. 写一个程序，实现功能：读入一个中缀表达式并写出与其等价的后缀表达式。
3. 写一个程序，实现功能：读入一个JVM指令序列，确定执行该指令序列可能导致的最大堆栈高度。从空堆栈开始。
4. 写一个程序，实现功能：读入一个JVM指令序列，确定其中是否有指令试图执行从空堆栈弹出的动作。(注意，这实际上是真实系统的验证器要完成的任务之一)。

第3章　用jasmin进行汇编语言编程

3.1　Java编程系统

就像在第1章所讨论过的，为什么一个用Java写的程序要在JVM上运行，或者为什么一个JVM程序不能用像C++这样的高级语言写成，在理论上都是没有道理的。在实际上，两者之间存在很强的关联，而且Java语言的设计已经强烈地影响了Java虚拟机和针对机器的汇编器设计。

特别是，Java强有力地支持和鼓励称为面向对象程序设计（object-oriented programming，OOP）的一种特定的程序设计风格。顾名思义，OOP是一种关注于对象的编程技术，对象就是这个世界上独立的活动元素（或者说是这个世界的一个模型），每个对象有其自身的一组可执行（或者说可于其上执行）的动作。对象又可归组成类（class），类是相似类型对象的集合，可根据其类型共享某些特性。例如，在真实世界中，“汽车”是一个自然的类，几乎很少有例外，所有的汽车都共享某些特性：它们都有方向盘、油门、刹车以及车前灯。它们也共享某些动作：可通过转动方向盘将车左转或右转，通过踩刹车使车减慢，或者由于油耗尽而使车完全停下。更着重地说，如果某人说他刚买了一辆新车，你可以假设这辆车有方向盘、刹车、油箱等等。

Java是通过将所有函数附于类并将所有可执行程序代码存储成单独的（并且可分离的）“类文件”来支持这种编程风格的。这些类文件与Linux的可执行文件或者Windows的.EXE文件有相当紧密的对应关系，区别之处是类文件无须保证其自身是功能完全的程序。相反，一个类文件只包含特定类的操作所必需的那些函数。如果它还依赖于另一种对象的性质和函数，则这些性质和函数就存储在一个对应于该对象的类中。

Java类文件与典型可执行文件的另一个主要差别是Java类文件在不同机器类型间是可移植的。故此，它不是用宿主机的机器语言编写的，而是用JVM的机器语言编写。这种代码称为字节码（bytecode）以表明它不依赖于任何特定的机器。由此，在任何机器上编译的任何类文件可自由拷贝到任何其他的机器上并仍能运行。从技术上说，JVM字节码只运行在JVM的一个副本上（就像Windows可执行文件通常只运行在一个Windows计算机上一样），但通过软件，JVM在几乎每个硬件平台上都能运行。

当一个字节码文件要执行时，要求计算机运行一个特殊的程序将类从磁盘加载（load）到计算机的存储器。此外，这时计算机通常还执行其他一些动作，比如加载支持当前类的其他类，验证该类及方法在结构上的完备性和安全性，将静态和类级别的变量初始化为适当的值。幸运的是，从用户或程序员的角度看，这些步骤都内置于JVM的实现中，所以用户不需要做任何事情。

这样，运行Java程序就是3个步骤的过程。在编写完程序源码后，必须将其编译或转换成一个类文件。然后用户必须创建一个（软件的）JVM实例用以执行字节码。做这件事情不同系统所用的具体命令不同。例如，在一个典型的Linux系统中，执行JVM的命令如下所示：

```
java TheClassOfInterest
```

这个命令就会去寻找名为TheClassOfInterest.class的文件，将其装入并进行验证和初始化。然后，该命令还将在这个类中寻找一个名为main()的方法，并试图调用这个方法。由于这个原因，任何独立的Java应用类必须包含一个“main”方法。在很多其他的系统上（例如Windows和MacOS），只要点击与.class文件对应的图标就会启动一个JVM并运行类文件。此外，某些种类文件可直接从Web浏览器（如微软的Internet Explorer或网景Navigator）来运行。

但是，这些程序实际上都不是在运行Java程序，而只是在运行JVM字节码。有许多编译器能将高级Java代码转换成JVM字节码，并且并不奇怪，也有程序能将不是Java的语言转换成JVM字节码。这里，我们将关注一种特定种类的语言，在这种语言中，每条字节码语句唯一地与程序源代码的单条语句相关联。如在第1章所讨论的那样，这种语言（即源代码语句与机器指令之间有1对1的关系）通常称为汇编语言（assembly language），而将一种语言转换成另一种语言的程序称为汇编器（assembler）。(对高级语言进行相应转换的程序则通常称为编译器（compiler）。)

3.2 使用汇编器

3.2.1 汇编器

正如所料，汇编语言与机器代码之间的转换是相当直截了当的。相应地,程序本身也相当容易写。汇编器的任务非常简单，所以对汇编器我们通常有若干种选择，而且这些选择之间的差异非常细微。Sun还没有为JVM确立一个官方的、标准的汇编器，所以本书中的例子程序都是针对jasmin汇编器所写的。这个程序在1996年由纽约大学媒体研究实验室的Jon Meyer和Troy Downing编写⊖。程序可从http://jasmin.sourceforge.net免费下载，并在事实上已经成为针对JVM的汇编语言的标准格式。jasmin程序还可以从本书的相关网站http:// prenhall.com/juola得到。这样，首要的步骤就是在你工作的机器上获得和安装jasmin。由于“Java虚拟机汇编语言”过于冗长难念，为简单起见，我们也将这个语言称为jasmin。

3.2.2 运行一个程序

为了执行图1-19中的程序（在这里我们将它复制到图3-1中），该程序必须首先被输入成机器可读的形式（如一个文本文件）。可以使用任何编辑程序做这件事情，从像Notepad这样的简单编辑器到复杂而且功能全面的出版用软件包都可以。但要记住，汇编器几乎不会处理各种奇特的格式和字体变化，所以要将程序存储成纯文本。根据著者惯例，用jasmin写的程序通常用.j扩展名存储，所以上面的程序就应以jasminExample.j存储到磁盘。

为了运行这个程序，就必须遵循与执行Java程序相同的步骤。在程序用文本格式编写完后，首先必须将其从人可读的jasmin语法转换成JVM机器码。其次，必须运行JVM（Java run-time engine）以使JVM码可执行。首先（对于适当配置的Linux机器），只要在适当的命令提示符后键入：

```
jasmin jasminExample.j
```

⊖ Jon Meyer的网站是http://www.cybergrain.com/(在我写这本书期间)。Meyer和Downing还有一本描述JVM和jasmin的极好的书（不幸绝版了）Meyer, J. & T. Downing. (1997). *Java Virtual Michine*. Cambridge, MA:O'Reilly。

```
; 定义相关的类文件为jasminExample.class
.class public jasminExample
; 定义jasminExample为Object的子类
.super java/lang/Object

; 创建对象所需的样板步骤
.method public <init>()V
     aload_0

     invokespecial java/lang/Object/<init>()V
     return
.end method

.method public static main([Ljava/lang/String];)V
     ; 对于System.out和字符串，需要两个堆栈元素
     .limit stack 2

     ; 寻找System.out（一个PrintStream类型的对象）
     ; 并将其放入堆栈
     getstatic java/lang/System/out Ljava/io/PrintStream;

     ; 寻找要打印的串（字符）
     ; 并将其放入堆栈
     ldc "This is a sample program."

     ; 调用PrintStream/println方法
     invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

     ; ...大功告成!
     return
.end method
```

图3-1 JVM汇编语言的样例程序

这将执行汇编器（就是在该文件上运行jasmin，产生一个名字为jasminExample.class的新文件，该文件包含了JVM可执行码）。

这个.class文件是Java运行时系统的一个标准部分，可用任何通常的方式来运行。最简单的方式就是键入：

```
java jasminExample
```

这将创建一个JVM进程，并在这个虚拟机上执行jasminExample.class文件（明确地说就是定义在该文件中的main方法）。

这种或类似的进程对于大多数机器和本书的所有例子都能行得通（当然那些故意含有错误的例子除外）。

3.2.3 显示到控制台还是显示到窗口

运行3.2.2节描述的程序使用了一个相当老式和风格不入流的交互界面。大多数现代的程序员倾向于使用窗口界面并通过鼠标点击来与计算机交互，而不愿使用命令行和基于文本的接口。Java流行的主要原因是其为窗口化和网络化应用提供了广泛支持。然而，这些应用与外部世界交互的方式存在微小的差异。

Java的最流行的一种Web应用称为applet。顾名思义，这是一种小的、相当轻量级的应用，

是特别为Web浏览器互操作而设计的，因而，所有主要的浏览器都支持那些结合了JVM applet的Web页。图3-2显示了一个非常简单的Web页的例子，其中唯一的内容就是一个<APPLET>标志。这个页的效果是：当它被显示在一个浏览器上时，浏览器将下载和运行这个applet。运行这个applet的具体方法是不同的。浏览器不是调用“main”方法，而是调用一个“paint”方法作为代码的起点。此外，输出指令（比如println）被图形专用指令（比如drawstring，该指令不仅接受要显示的串，而且还接受该串显示在窗口中的位置之类的参数）代替。

```
<HTML>
<HEAD> <TITLE>A Sample JVM Applet</TITLE> </HEAD>
<BODY>
<APPLET code = "jasminAppletExample" width = 300 height = 100>
</APPLET>
</BODY>
</HTML>
```

图3-2 调用applet的Web页

applet编程是一门精细的艺术，需要applet专用函数的一些知识。利用这些函数，一个熟练的程序员能创建精致的图画，或者所需的任何字体、大小、形状和方向的文本。如图3-3所示，无论是写成applet，还是输出到窗口，或者是写成向控制台输出的独立应用，jasmin程序的总体结构不会改变很多。

像以前一样，程序（jasminAppletExample.j）将用jasmin程序汇编，产生一个类文件：

```
jasmin jasminAppletExample.j
```

一旦创建了类文件jasminAppletExample.class，只要打开图3-2所示的Web页就可运行applet。这可在任何Web浏览器（例如Internet Explorer或者Netscape）中打开，或者用由Sun公司连同Java系统一起提供的特殊程序如appletviewer。利用这种技术，Java（和jasmin）程序就可作为可执行代码，并由任何地方机器上的JVM所使用。

3.2.4 使用System.out和System.in

当用任何汇编语言编程时，让计算机读写数据属于最具挑战性的任务之一。计算机读写问题与计算机和I/O外设进行交互等更一般的问题相关。由于外设类型繁多（从网络读与从键盘读有很大的差异），而且更令人烦恼的是，即使给定了外设类型，其中的差异也很大（你的键盘有没有数字键区？），所以每个设备就不得不用各自不同的方法来处理。

Java的类系统稍微缓和了这个问题。由于所有的车都有方向盘并以相同方式工作，所以人们可以驾驶不熟悉的车。同样，Java定义了PrintStream类，该类包括了名为print和println的方法。JVM总是定义一个特殊的PrintStream，名为System.out，它附加于一个默认的能打印的设备上。

这就提供了一个相对简单的方法，可以通过print或println来产生输出。在图3-1的简单例子中已经演示过打印String类型，还可扩展用于打印该方法支持的任何类型（要做一些改动）。这里给出的必要步骤大多没有做解释，你不必现在就理解它们。要全面理解这些步骤，就要求对JVM类型和类系统以及它们是如何表示的等问题有更深入的研究。我们将在第10章再返回来更详细地讨论这个简单例子。

首先，System.out对象必须从其在系统中的静态和不改变的位置压入到堆栈。

```
getstatic java/lang/System/out Ljava/io/PrintStream;
```

其次，要打印的数据必须采用第2章介绍的常用方式装入到堆栈。

```
; 定义jasminAppletExample作为Applet的子类
.class public jasminAppletExample
.super java/applet/Applet

; 创建applet所需的样板程序
; 注意与前面例子中Object创建的相似性
.method public <init>()V
   aload_0
   ; 这不是一个Object，所以我们必须调用Applet <init>
   invokespecial java/applet/Applet/<init>()V
   return
.end method


; 注意applet开始于paint()方法而不是main()
; 还要注意有微妙差异的定义
.method public paint(Ljava/awt/Graphics;)V
   ; 我们需要4个堆栈元素
   ; Graphics对象
   ; 要打印的串
   ; 打印位置的x坐标
   ; 打印位置的y坐标
   .limit stack 4

   ; 存储在局部变量#1的Graphics对象
   .limit locals 2

   ; 这个样板程序有点不寻常，因为在applet中绘制文本要比在System.out上难
   ; 装入4个参数
   aload_1          ; 这是作为参数传递的Graphics对象
   ldc "This is a sample applet" ; 要打印的串
   bipush 30        ; 打印该串的坐标
   bipush 50

   ; 调用drawString方法
   invokevirtual java/awt/Graphics/drawString(Ljava/lang/String;II)V

   ; ...大功告成!
   return
.end method
```

图3-3　jasmin的一个样例applet

```
iload_2     ; 作为整数装入局部变量#2!
```

第三，必须调用println方法，包括在第二步中压入到堆栈的类型表示。由于语句iload_2压入了一个整数，因此命令就是：

```
invokevirtual java/io/PrintStream/println(I)V;
```

如果我们压入的是浮点数（可用fload_2），命令就要修改成用F替代I，如下所示：

```
invokevirtual java/io/PrintStream/println(F)V;
```

要打印一个字符串，就需要样例程序中使用的那种复杂的语句。

```
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V;
```

如果你觉得糊涂，先不要担心。类和类的调用将在第10章详细讨论。现在这些语句可看作

一种合法的样板或魔术。每当你要产生输出时，只要使用适当的版本就行了。可以用一组类似但更复杂的样板程序从类似构造的System.in对象得到输入，但这个讨论将推迟到我们对类的一般概念有了更好的理解后再展开。现在，理解基本语句在开发jasmin程序中更为重要。

有了最新版本的Java（Java 1.5），采用新定义的类如Scanner和Formatter，以及新的结构化的用于格式化输出的printf方法，就可以得到一个全新的进行控制台输入和输出的方法。从底层汇编语言的角度看，这些都被看作是可调用的新的函数/方法。它们不会牵涉任何技术上的根本改变。

3.3 汇编语言语句类型

3.3.1 指令和注释

汇编语言语句可粗略地分成三种类型（特别是，对于jasmin语句也是这样的）。第一种类型是指令（instruction），直接对应于计算机的机器语言或字节码的指令。在很多情况下，这些是通过查找汇编器中存储的表而生成的。对应于助记符iconst_0的字节码就是位模式0x03。此外，大多数汇编器允许使用注释，程序员能够插入提示和设计记录，从而帮助他们自己以后能理解现在正在做的事情。在jasmin的一行中，任何在分号（；）以后的部分都是注释，所以在图3-1的前两行都是注释。汇编器忽略注释，就像它们不存在一样，所以在汇编过程中注释的内容被跳过，但在源程序中这些内容是可见的（便于人阅读，比如便于正在给你的程序打分的教授来阅读）。

在众多汇编器中，jasmin程序在允许程序员自由使用编程格式方面有点与众不同。汇编语言程序的语句通常有一个非常老套和不灵活的格式，比如像下面这样：

```
标号:
            助记符      参数      ; 注释
```

助记符/参数组合在前面已经见到，比如在像iload 2之类语句中。根据助记符类型的不同，可以有任意数量的参数，不过零个、一个及两个参数是最常见的情况。标号将在第4章详细讨论。现在只需指出它标记了程序的一个部分，使得你能返回并重复执行一段代码。最后，这个老套的语句包含一个注释。从技术上说，计算机永远不会要求你对程序加注释。另一方面，你的老师几乎总会要求你加注释，而好的编程习惯也要求你加注释。特别是，多数的汇编语言标准通常要求每行至少有一个注释。

本书和jasmin一样，对于注释持有稍微不同的观点。因为jasmin程序中用的很多参数都是长整数，尤其是字符串参数以及像系统输出这样的标准对象地址，这样，在一行中就没有地方再放相关的注释了。对于每行一个注释这个标准，一个更严重的问题是可能会鼓励拙劣而无有用信息的注释。

作为一个例子，请见下面这一行语句：

```
bipush    5    ; 将整数5装入到堆栈
```

这个注释几乎没有告诉程序员任何东西。毕竟语句bipush 5就是“将整数值5装入到堆栈上”的意思。在读到这条语句时，任何程序员即使独自在看也知道这是什么意思。为了理解程序，程序员可能还需要了解更宽泛问题的答案。为什么在这个步骤中要将5这个特定的值压入到堆栈中（而且为什么要作为整数装入）呢？通过强调语句在更大范围的作用以及成块或成组语句的意义，就能使注释更加有用和富含信息。

```
bipush    5    ; 装入五边形的边数来度量
```

或者，甚至可以写作：

```
bipush  5    ; 装入五边形的边数来计算周长
             ;
bipush  8    ; 依次装入边1～5的长度
bipush 13
bipush  9
bipush  7
bipush  2
             ; 现在将各个边累加在一起
```

为此，建议不要盲目遵从“每行一注释”这样的人为标准，而是理性地遵从“注释应该有解释性”的思想。而且，汇编语言程序尤其需要很多解释。

3.3.2 汇编指令

第三类语句称为汇编指令（directive），是对汇编器本身的指令，告诉它如何执行其任务。在jasmin中，大多数的汇编指令都用一个句点（.）开头，如样例程序中第三行（即第一个非注释行）所示。例如，汇编指令（.class）的功能就是通知汇编器：这个文件定义了一个名为jasminExample的类，因此，要创建的类文件名字就是jasminExample.class。这并不直接影响将程序转换成字节码的过程（且不对应于任何字节码指令），但它直接通知jasmin如何与计算机的其余部分、磁盘以及操作系统相互作用。

在这一点上很多汇编指令可能没有明确的含义。这是因为JVM和类文件本身都直接捆绑到面向对象结构和类层次。例如，所有类必须绑定到类层次，尤其必须是其他类的子类。（例如，梅赛德斯Mercedes是汽车car的一个子类，而汽车是运输工具vehicle的一个子类，等等。）Java语言是通过这样的方法来实施这一规定的：对于任何没有明确提及其在层次中的关系的类，默认它是Object（java/lang/Object）的子类。jasmin汇编器强制实施类似的要求，就是任何jasmin定义的类必须包含一个.super汇编指令，以定义这个新的类是哪个超类的子类。在各个jasmin程序中，程序员通常可简单地拷贝下面一行而不会有什么损害：

```
.super java/lang/Object
```

其他的汇编指令（.method、.end method）用于定义该类与所有其他对象的相互作用。特别是，JVM所实施的面向对象编程模型要求，从类外部对函数的调用要明确地定义为“public方法”，并且强烈地鼓励所有函数都这样定义。这种定义的细节将在第10章做非常详细的讨论。

3.3.3 资源汇编指令

从jasmin程序员的角度看，最重要的汇编指令就是.limit。这个汇编指令用来定义限制，或者展开来说就是在一个方法中进行计算可得到多少资源。这是JVM（作为虚拟机）的状态中独一无二而且非常强大的方面，因为方法可按其所需使用任意数量的资源。

特别是，一个典型的基于堆栈的微处理器或控制器（例如老式的英特尔8087数学协处理器芯片，后来被合并到80486及后续机型成为主CPU的组成部分）就只拥有较少的元件（在这种情况下是8个）。一个需要多于8个元件的计算就需要将某些值存储到CPU堆栈以及如主存储器等其他一些地方。程序员必须确保数据按需要移入和移出存储器，失效的代价通常是程序出故障或整体功能不正常。增加CPU内可利用堆栈空间的数量可解决这个问题，但却会使每个CPU芯片更大、更热、更费功耗而且更昂贵。此外，改变不同型号芯片之间的基本参数会引入不兼容性，使得新的程序不能在老机器上运行。

在JVM上相应解决方法的特色是干净利落。可以使用汇编指令：

```
.limit stack 14
```

这条汇编指令作为一条语句直接放在所定义方法的内部（使用.method汇编指令），其功能是将堆栈的最大尺寸设置为14个整数或浮点大小的元素。（它也能存储7个长整数或双精度大小的元素，或者5个长整数/双精度和4个整数/浮点元素，或者任何适当的组合。）类似地，在一个方法中（int大小的）局部变量的最大数量可用相关的汇编指令设置为12：

```
.limit locals 12
```

如果这两个汇编指令中的任何一个被忽略，则应用一个缺省的限制值，这个值足够用于单个整数或单个浮点数，却不足以用于更大的类型。

3.4 例子：随机数生成

3.4.1 生成伪随机数

一个常见但在数学上很复杂的任务（而且是计算机经常要执行的任务）是生成貌似“随机”的数。例如，在计算机游戏中，可能有必要将一副牌洗成貌似随机的次序。由于在当今计算机硬件方面有几个根本的限制，计算机实际上没有能力生成随机数（在统计学家们所坚持的严格意义上）。作为替代，计算机生成的是确定性的伪随机数据，这些数据虽然在技术上来看可预测，但看起来似乎是不可预测的。

我们在这里关注的是生成均匀分布在0~n这一范围内的随机整数。如果由于某种原因用户希望生成随机的浮点数，可以通过简单地将随机整数除以$n+1$得到。如果n足够大，这种方法就对实数在区间[0,1)上的均匀分布给出了一个很好的近似。（例如，如果n是999，浮点数就是集合{0.000,0.001,0.002,...,0.999}中的一个。如果n是10亿，最终的数看起来就非常随机。）

从数学上看，计算机接收一个给定的数（即种子）并返回一个相关但貌似不可预测的数。通常的做法就是采用所谓的线性同余数生成器（linear congruential generator）。采用这种方法，对于特定的a、c及m值返回的数由下面形式的公式生成：

$$newvalue = (a \cdot oldvalue + c) \bmod m$$

例如，因为计算mod m给出的最大结果是$m-1$，所以参数m决定了返回的随机数的最大值，由此$n=m-1$。在选择最佳a和c的取值方面有很多理论研究，所以过多在这方面探究就离题太远了。*oldvalue*的值必须在每次运行生成器时重新选择，因为*newvalue*的值严格依赖于它。这个生成器可重复使用来生成一序列的（伪）随机数，所以对每个程序只需要确定一次种子。初始种子的典型来源包括（对程序来说）真正的随机值，如一天的当前时间、进程的ID号、鼠标的最近运动等等。

3.4.2 在JVM上实现

为了在JVM上实现这个算法，就要做出一些设计决策。基于实际的原因，*oldvalue*的值可能要存储在局部变量中，因为其值在每次调用时会发生变化，但是a、c及m的值可以作为常数存储和操作。为了简便起见，*oldvalue*和*newvalue*的值都被存储成整数作为一个堆栈元素，但那些中间值，尤其是a和c是大数时，就可能使单个堆栈元素溢出，因此必须将它们存成长整数类型。无需多做解释，我们采用可存储成（有符号）整数的最大质数（2 147 483 647）作为m的值，并选择质数$2^{16}+1$（=65537）作为a的值，5作为c的值。

计算本身简单明了。上面的表达式

$$(a \cdot oldvalue+c) \bmod m$$

可用JVM（逆波兰法）表示成：

$$a\ oldvalue \cdot c + m \bmod$$

因此，适合的JVM指令如下：

```
; 计算a*oldvalue
ldc2_w  65537           ; a为2^16+1，存储成长整数
iload_1                 ; oldvalue（假设）被存储成局部变量#1
i2l                     ; 将oldvalue转换成长整数
lmul                    ; 做乘法

; 加c
ldc2_w  5               ; c是5，存储成长整数
ladd                    ; 加入到a*oldvalue

                        ; 得到mod m的余数
ldc2_w  2147483647      ; 装入m的值
lrem                    ; 取模（求余数）
l2i                     ; 转换回整数

; newvalue现在就留在堆栈顶部
```

补充资料

参数传递、局部变量以及堆栈

大多数程序要求输入是有用的。事实上，大多数函数和方法都要求输入是有用的。让方法获得信息的标准方式是通过参数传递。例如，定义正弦通常要有一个形式参数（formal parameter）。当使用正弦函数时，相应的实际参数（actual parameter）就被传递到函数并用于计算。

JVM用一种相当奇特的方式来处理这个过程。在传统基于芯片的体系结构中，计算机在内存中采用一个共享的“堆栈”来分隔不同程序或函数所使用的存储区域。与之相反，JVM对每个方法提供了唯一的私用堆栈。这就避免了一个方法闯入并毁坏程序的其他部分所独有的数据，从而极大地增强了安全性，却使一个函数向另一个函数传递数据变得困难。作为替代的方法是，当一个函数/方法被调用时，参数被（JVM）置于方法所能得到的局部变量中。一般来说，第一个参数被置于局部变量#1，第二个参数被置于局部变量#2，依此类推。

对这个一般规则有3个例外。首先，如果参数太大，不能装入到单个堆栈元素（长整数或双精度数）中，就要置于两个连续的元素中（所有后来的元素就要多下移一个元素位置）。其次，这个规则没有考虑局部变量#0。通常对于实例方法（instance method），对当前对象的引用将用#0传递。定义为静态（static）的变量没有当前对象，因此，将第一个参数传递到局部变量#1、#0，依此类推。

最后，Java 1.5定义了一个新的参数传递方法，在参数数量变化的时候可以采用。在这种情况下，数量变化的参数将被转换成数组并作为单个数组参数传递（可能在#1中）。被调用的方法负责确定实际上要传递多少参数，并正确地对其进行操作。

在不是JVM的机器上使用堆栈进行参数传递的方法将在后续章节讨论特定机器时详细描述。

通过观察，在这些计算中所需的最大堆栈深度是两个长尺寸（双精度数）堆栈元素，所以这可在堆栈限制为4或更多时用任何方法执行。类似地，给出的代码假设*oldvalue*存储在局部变量#1中。因为变量从0开始编号，这就意味着程序需要两个局部变量。（*oldvalue*要存储在变量#1中的原因是：在某些情况下，局部变量#0由Java类环境所保留。）

为了使这个程序正确运行，包含这个代码的方法需要两个汇编指令：

```
.limit stack 4
.limit locals 2
```

这个代码有几个变型，它们也能工作。像大多数的编程问题一样，存在若干个正确的解决方案。最明显的是，两条汇编指令可颠倒次序，先定义局部变量再定义堆栈大小。更复杂的变化是采用不同的操作次序来进行计算，也许是：先压入c，做乘法，然后做加法。从技术上说，这就是实现等价但不同的逆波兰表达式：

$$c\ a\ oldvalue\ \cdot\ +m\ \mathrm{mod}$$

如果选择了这种实现，则堆栈的最大深度就是3个长元素，需要一个`.limit stack 6`汇编指令。

类似地，执行很多小的步骤也有一些等价的方式。如果不是（用`ldc2_w 5`）将值5作为长整数直接压入，程序员也可以（用`iconst_5`）将值5作为整数压入，然后再（用`i2l`）将其转换成长整数。这会将一条指令换成两条，但这两条替换成的指令可能更短且执行更快。然而，像这样的微小改变很少能对程序的大小和速度有重大的影响。更常见的情况是，这些只是执行相同任务的不同方式，并有可能使新手程序员不知所措，因为他往往期望对一个给定问题只有一个解。

3.4.3 另一种实现

不但上面提到的随机数生成问题有多种解决方案，而且有很多不同的算法和参数能解决这个问题。仔细地考察JVM的表示方案，就可以得出优化的而且更复杂的随机数生成器。特别是，由于涉及整数的数学运算总是采用32位的量来执行，将数对2^{32}取模就是自动进行的。通过将m（隐含地）置为2^{32}，程序员就能避免涉及取模这部分计算。而且，如果所有的计算都隐含地以这种模的形式来完成，就不需要使用长尺寸的存储器或堆栈元素。

采用这种模能产生好的随机数生成器的一组数是：置a为69069，置c为0。（这些数实际上是由研究者George Marsaglia所提出的“极高超”（Super-Duper）生成器的一部分。）特别是将c置为0还会简化代码，因为无需做加法。所得到的代码将短小、简单而且优雅。

```
; 计算  a * oldvalue
ldc2_w   69069       ; 为Super-Duper提出的，作为a的值
iload_1              ; 假设oldvalue存储为局部变量1
imul                 ; 做乘法（隐含地取2^32模）

        ; newvalue现在就在堆栈顶部
```

那么，哪个随机数生成器更好呢？比较生成器的优劣可能是非常困难，并可能涉及相当高强度的统计计算。此外，根据你的应用不同，线性同余法一般会有一些不好的性质。而且，根据使用生成器的方式不同，结果的某些位可能会比其他位更随机。例如，如果*oldvalue*是奇数，第二个生成器就会总是生成奇数，否则就会总是生成偶数。只使用高序字比只使用低序字会给出好得多的结果。对这些生成器所产生数的质量比较时，最容易的方式是将它们都在计算机上实现，且都运行几千、几百万或者几十亿次，并根据所期望的用途接受统计检验。

从速度的角度看（而且更重要地，从计算机组成和汇编语言课程的角度看），显然第二种

随机数生成器会运行得更快。不仅是因为它涉及了更少的操作，而且因为这些操作本身会按整数运算来运行，因此比第一个生成器的长操作速度快。

3.4.4 与Java类交互

（这一节可跳过而不影响连续性，假设你具备一些Java编程知识。）

既然这样，为什么要假设种子（*oldvalue*）存储在局部变量#1中呢？这直接关系到方法如何用Java实现以及JVM如何处理类之间的对象和方法的相互作用。特别是，一旦一个对象的方法被调用，JVM就以局部变量#0传递对象本身（作为引用类型变量来传递，因而在操作助记符中就以a打头），而各种方法参数则以局部变量#1、#2、#3等等依次传递（根据需要可以有任意数量的变量/参数）。

为了正确地运行，上面描述的第二个生成器就要置于一个至少有两个堆栈元素和至少两个局部变量（一个用于对象，一个用于种子值）的方法中。一个完整的jasmin样例程序见图3-4，该程序定义了一个特殊的类（jrandGenerator.class）和两个方法，一个用于对象创建，另一个用于通过上述的第二种方法生成随机数。

```
; 定义jrandGenerator作为Object的一个子类
; 定义与此相关的类文件为jrandGenerator.class
.class public jrandGenerator
.super java/lang/Object

; 标准步骤，与前面一样
.method public <init>()V
   aload_0

   invokespecial java/lang/Object/<init>()V
   return
.end method

; 定义一个Generate()方法，接受整数并返回整数
.method public Generate(I)I
   ; 为计算，我们需要两个堆栈元素
   .limit stack 2

   ; 我们还需要两个局部变量，其中#1保存参数
   ; 由于这是一个标准的方法，#0会由Java本身设置
   .limit locals 2

   ; 计算a*old_value（并存储在堆栈顶部）
   ldc     69069          ; 作为a的值
   iload_1                ; old_value假设存储为局部变量1
   imul                   ; 做乘法（隐含地完成mod 2^32运算）

   ; new_value被存储在堆栈顶部，作为整数返回
   ireturn

.end method
```

图3-4 jasmin中完整的随机数生成过程

该程序的结构与先前打印一个字符串的程序有紧密的对应关系。然而，与先前程序不同的是，没有定义main()方法（不要求jrandGenerator类是一个独立的程序）。它要求的是多个局部变量（定义在.limit汇编指令中），另外Generate方法的参数和返回类型已经改变，以反

映其作为随机数生成器的用途。

当用jasmin程序进行汇编时，结果将是一个名为jrandGenerator.class的Java类文件。在Java编程环境中，这个类的对象与其他任何对象一样创建和使用，如图3-5所示。这个简单的程序只是创建了一个jrandGenerator的实例，并在其上连续快速地调用10次Generate方法，因而生成了10个随机数。

```
public class jrandExample {
      public static void main(String args[]) {
            int i;
            int old_value = 1;
            jrandGenerator g = new jrandGenerator();

            for (i=0;i<10;i++) {
                  old_value = g.Generate(old_value);
                  System.out.println("Generated: " + old_value);
            }
            return;
      }
}
```

图3-5 调用jrandGenerator的Java程序

类似地可写一个程序生成1千万个随机数，或者调用一个不同的生成器（实现前述的第一个随机数生成方法）。

3.5 本章回顾

- JVM与高级编程语言Java一起设计，因此支持一种类似的面向对象的程序设计（OOP）风格。虽然并非必须在jasmin编程中使用OOP，但这通常是一种好的想法。
- Java程序和jasmin程序必须都转换成.class文件后才能由JVM执行。
- 将jasmin程序转换成（本书中用到的）类文件的命令通常命名为jasmin。在写这本书的时候，可以在Jon Meyer的Web站点或相关的Web站点http://prenhall. com/ juola上免费得到。
- 由于有大量不同的设备，所以输入和输出是典型的困难问题。Java和JVM通过使用类系统来简化这个问题。通过熟记三条正确的jasmin语句，程序员就能在任何时间将（任何类型）数据发送到标准输出。
- 汇编语言语句可分成3种主要的类型：指令（要被转换成字节码机器指令）、注释（被计算机忽略）以及汇编指令（要影响转换/汇编过程本身）。
- 汇编指令用于定义类文件如何嵌入到标准Java类层次中。
- 汇编指令，尤其是.limit汇编指令，也用于控制方法和函数所能得到的资源数量。特别是.limit stack X要设置最大堆栈尺寸，而.limit locals Y要设置局部变量的最大数量。

3.6 习题

1. 编译器和汇编器的区别是什么？
2. 列出至少5条只需一个参数的指令（助记符）。

3. 计算机如何辨识出一个给定的行包含的是一个汇编命令、一个注释还是一条指令？
4. 如果你忘记了`.limit`汇编命令，你的`jasmin`程序还能工作吗？

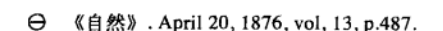
3.7 编程习题

1. 写一个`jasmin`程序，将下面的诗显示在一个Web页面上。

    There once wad a lady named Nan
    Whose limericks never would scan
    When she was asked why
    She replied with a sigh.
    “It's because I always try to put as many syllables into the last line as I possibly can”
2. 写一个`jasmin`程序，显示如下所示的由大写O组成的三角图案。



3. 写一个`jasmin`程序，用以下格式显示今天的日期：Today is Monday, 9/19/2008。
4. 写一个`jasmin`程序，计算并显示在下面情形下我应得的报酬：这个月，对于每小时薪金25.00美元的工作我的工作时间是80个小时，对于每小时薪金15.50美元的工作我的工作时间是40个小时，对于每小时薪金35.00美元的工作我的工作时间是45个小时。
5. 波音777-300飞机的最大载客量为386。写一个程序，确定为了承载*N*个人做环球旅行，你需要包租多少架飞机。你可以在程序中用一个特定的*N*值。（注意：飞机按架包租，不允许包租一架飞机的五分之三。）
6. 2月29日这一日期每4年只出现一次。例如，在2000年、2004年及2008年出现。如果我的一个朋友出生于1980年2月29日，而当前的年份是Y_1，写一个程序，告诉我他实际上已经有多少年能庆祝生日。（你可以假设他今年没能庆祝生日。）
7. 与圣诞节不同（总是在12月25日），复活节每年的日期都不一样。《自然》[⊖]杂志的一位匿名记者发表了确定复活节日期的算法。（这个算法后来被米斯郡主教Samuel Butcher证明是正确的，因此被称为Butcher算法）。所有值都是整数，所有除法都是整数除，且mod的意思是整数取模（除法后的余数）：
    - 设*y*为相关年
    - 设*a*为*y* mod 19
    - 设*b*为*y*/100
    - 设*c*为*y* mod 100
    - 设*d*为*b*/4
    - 设*e*为*b* mod 4
    - 设*f*为(*b*+8)/25
    - 设*g*为(*b*−*f*+1)/3



⊖ 《自然》. April 20, 1876, vol, 13, p.487.

- 设h为$(19 \cdot a+b-d-g+15) \bmod 30$
- 设i为$c/4$
- 设k为$c \bmod 4$
- 设l为$(32+2 \cdot e+2 \cdot i-h-k) \bmod 7$
- 设m为$(a+11 \cdot h+22 \cdot l)/451$
- 设p为$(h+l-7 \cdot m+114) \bmod 31$
- 复活节月是$(h+l-7 \cdot m+114)/31$.(3=3月, 4=4月).
- 复活节日是$p+1$

实现这个算法，确定下10个复活节的日期。

第4章 控制结构

4.1 他们教给你的都是错误的

4.1.1 再谈取指－执行

在第2章和第3章中，我们探讨了如何用JVM的基于堆栈的计算写出复杂的数学表达式并对其进行求值。通过将参数压入计算堆栈并执行适当的基本操作序列，就能使计算机或多或少地按人的吩咐进行工作。从实用化的观点看，计算机的真正优势是其拥有这样的能力，即它能不厌其烦或正确无误地反复执行任务。

回顾JVM的表示结构，不难看出如何使计算机反复执行相同的代码模块。记住，程序代码是由连续的机器指令组成的，在计算机的存储器中存储为连续的元素。为了执行一个特定的语句，计算机首先从存储器中取出当前指令，解释并执行该指令，然后更新“当前指令”的标志。正式地说，当前指令就是一个存储在程序计数器（PC）中的数，指向当前方法字节代码的位置。每次取指－执行周期发生，存储在PC中的值就增加1个或更多字节，使得它指向要执行的下一条指令。

为什么是1个“或更多” 字节呢？难道PC不应该每次增加1吗？事实上不是这样，因为一些操作需要多个字节来定义。例如，像irem这样的基本算术操作只需要1个字节来定义。然而，像bipush这样的操作用1个字节就不够了。bipush规定要将1个字节压入到堆栈（并被提升成为32位整数），但它本身并不规定要压入哪个字节。一旦使用了这条指令，bipush的操作代码（0x10）后面就要跟一个要压入的字节。正如所料，sipush指令（0x11）后面跟的就不是1个而是2个要压入的字节（是一个短整数）。类似地，iload指令后跟1个或2个字节，描述了要装入的局部变量。相比之下，iload_1快捷操作（0x1B）自动地装入局部变量#1，这样就能用1个字节来表示。

由于操作大小是变化的，为了取出全部的指令及其所有参数，取指－执行周期就需要足够聪明，能一次或依次取出可能的若干个字节。PC的更新必须反映取出指令的大小。一旦这些困难得到处理，将PC设置为适当位置就能使得JVM自动地执行指令序列。因此，如果有强制PC包含某个特定值的方法，就能使计算机反复执行那个代码块。通过控制PC，就能直接控制计算机做什么以及做多少次。

在后面的几个小节中，这种直接控制就等价于常遭蔑视的goto语句。常常告知并一直灌输学生们的是避免使用这种指令，因为与控制方便的程序块的相比它们会引入更多的错误。在高级语言中，学生们在学习编程方法时被告知要避免使用goto语句。在汇编语言一级，就赤手空拳了，我们能做的最好事情就是理解它们。

4.1.2 转移指令和标号

任何可能导致PC改变其值的语句通常称为“转移”指令。与正常的取指-执行周期不同，一条转移指令可能转到任何地方。为了定义目标位置，jasmin像大多数其他的汇编语言一样，允许单个指令接受标号（label），使其能作为个体被引用。不是所有的指令都会得到这样的标

号，但任何语句都能得到。要调整PC，可以使用适当的语句，然后给出目标指令的标号。

那么，一个转移语句是如何生成和存储的呢？更详细地说，就是什么是“标号”？标号如何被存储成位的序列，像一个数或机器指令那样吗？从程序员的角度看，标号就是一个独立的一行，可以保存或不保存任何指令，标号本身是一个标记了指令的字（由字母和数字组成，以字母开头，按惯例是大写字母，后跟一个冒号[:]）。要将控制传递到一个给定的位置，就要在一个适当的转移语句中使用标号（不加冒号）作为参数。例如：

```
goto Somewhere ；将PC的值设置为Somewhere的位置
```

4.1.3 结构化编程：转移一下注意力

从机器设计的观点看，控制PC的最简单的方式就是将其作为一个寄存器或局部变量来看待，并为之指定适当的值（见图4-1）。当一个特定的值被放入到PC时，机器将转移到那个位置并开始执行代码。

这当然就是无限度地滥用goto语句。

```
；做一些计算
；再做一些计算
goto ALabel；现在将控制直接传递到ALabel
；这条语句被跳过
；这条语句也被跳过
ALabel:
；但我们从这里的语句开始
；并以正常的方式继续运行
```

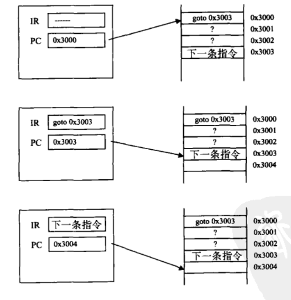


图4-1 执行了一条goto语句的取指-执行周期

根据现代程序设计的实践（大约1970年以来以及Dijkstra的非常有影响的工作），使用goto语句的编程遭到严厉的责难。一个原因就是无保护的goto可能是危险的。将一个随机的位置放入到PC将会导致计算机执行存储在那个位置的任何指令。如果那个位置是计算机代码，还算可以。如果那个位置恰好（比如）处于存放串变量的存储位置中间，计算机就会将串的各个字节

看成是程序代码并开始执行。（例如，在JVM中，ASCII字符65（A）对应于lstore_2，就会导致计算机试图从堆栈弹出一个长整数。如果堆栈顶部没有这样的长整数，就会引发程序崩溃）。

一个潜在的更严重的问题是带有goto语句的程序会引起混乱并因而易于出错。从作为goto目标的语句的角度看，在程序的形象结构、字节码级的程序语句顺序以及程序语句的执行顺序之间没有明显的关系。请看下面的简单语句序列：

```
        iload_1
        iload_2
Branch:
        iload_3
        iadd
        imul
```

该序列可能并不像不经心的读者所想的那样，因为代码可能通过一个goto语句转到第3个iload_N语句来执行。这样，存储在局部变量#3中的值被加入到（并被乘以）某个并不明确的值。在特别糟糕的情况下，控制结构可能是完全混乱的（通常被比喻成意大利式细面条），存在导致程序员混乱的所有常见问题。

由于这个原因，现代"结构化编程"推荐使用高阶控制结构，如块结构的循环和判定语句。然而，在机器代码级别，任何反映不同计算的变化必须通过对PC的变化表达出来，这样就要求有一个隐含的goto。

那么，为什么希望程序员不用goto语句来编程呢？结构化编程的思想不是完全避免使用goto，而是限制其在不会引起混乱的场合使用。特别是，程序应尽可能模块化（由逻辑上分开的代码片断组合而成，这些代码片断在概念上可作为单个操作看待）。这些模块应尽可能有单一的起始点和单一的退出点，理想情况是起始点和退出点分别在实际代码的顶端和底端。（这常常被规范化为"单入口/单出口"原则，作为结构化编程定义的组成部分）。高级语言经常通过其设计防止违反单入口/单出口规则。大体上，同样的原则能够也应该应用到汇编语言编程中。虽然语言本身会给你灵活性以至于做出非常愚蠢和混乱的事情，但训练有素的程序员会抵制这种诱惑。熟悉高级控制结构（如if语句和循环）的程序员会努力使用类似的易于理解的结构，甚至是在jasmin中，使得他们及其合作者能够准确地领会程序的意图。

4.1.4　高级控制结构及其等效结构

一个简单的例子有助于阐明这一点。图4-2和图4-3分别用Java/C++（还有很多其他语言）和Pascal给出类似的循环例子。两个例子实现的都是从100计数直到0，在每次循环都会做一些聪明的事情。做聪明事情的模块是用最高效的方式写成的连续一组机器指令。当装入了这组指令的首条指令的地址时，整个模块就将被执行。

```
for (i=100; i > 0; i--) {
      // 做100次聪明的事情
}
```

图4-2　用Java或C++写的循环示例

```
for i := 100 downto 1 begin
     { 做100次聪明的事情}                }
end
```

图4-3　用Pascal写的等价示例

为了执行这个模块若干次，计算机需要在每次模块的起始点决定该模块是否还需要再（至少）执行一次。对于图中循环的情况，决策很容易做出：如果计数器大于0，则模块应再次执行。在这种情况下，就转到模块的开始处（并递减计数器）；否则，如果计数器小于或等于0，就不再执行循环并转移到程序的其余部分。

非正式地，这可以用简单的pop-and-if->0-goto来表达。正式地，这个特定操作的助记符

是ifgt。将循环正式转换到jasmin，结果如图4-4所示。

```
          ldc 100            ; 装入整数100（循环次数）
          istore_1           ; 将索引作为整数存储到#1
LoopTop:
          ; 做一些特别聪明的事情
          ; 使用所需要的局部变量
          iload_1            ; 从#1重新装入循环索引
          iconst_m1          ; 装入-1，用于减法
          iadd               ; 循环索引递减
          istore_1           ; 存储...
          iload_1            ; ...并重新装入循环索引
          ifgt LoopTop       ; 如果堆栈顶部>0，就将LoopTop放入PC
                             ; 并重复
; 否则，不再循环
```

图4-4 一个用jasmin写的（几乎）等价的循环示例

实际上，这不是一个完全准确的转换。只要循环索引的初始值大于0，它就会正常运行。然而，如果程序员规定的循环起始值是负数（如for (i=-1; i>0; i++)），则用Java或C++写的循环将永远不会执行。jasmin版本仍能执行一次，因为在计算机有机会检测索引是否足够大之前，做聪明事情的计算已经执行了。更准确的转换需要更巧妙或者至少更有变化性的判定和goto语句。

4.2 goto的类型

4.2.1 无条件转移

最简单形式的goto就是无条件转移（unconditional branch），（就是goto语句），该语句总是简单地将控制转移给被指定为参数的标号。该语句本身能产生无限循环（永远运行的循环），而不是只运行一段时间然后在某些情况改变后就中止的循环。为此，程序员就需要条件转移（conditional branch），就是可能发生也可能不发生的转移。

4.2.2 条件转移

JVM支持6种基本的条件转移（有时称为“条件goto”），再加上若干种便捷操作和另外两个转移（将在后面与类/对象一起描述）。一个基本的条件转移是通过从堆栈中弹出一个整数，然后确定这个弹出的整数是大于、小于还是等于0。如果所期望的条件满足，控制就转移到指定的标号。否则，PC像往常一样递增并将传递到下一条语句。用3种可能的比较结果（大于0、小于0或等于0），就能指定所有7种有意义的组合。这些在表4-1中做了总结。注意goto语句不改变堆栈，而if??操作都弹出单个整数。

表4-1 jasmin中的条件和无条件转移操作

助记符	top>0	top=0	top<0	解释
ifeq		X		如果等于就goto
ifne	X		X	如果不等于就goto
iflt			X	如果小于就goto
ifge	X	X		如果大于或等于就goto
ifgt	X			如果大于就goto
ifle		X	X	如果小于或等于就goto
(goto)	X	X	X	（总是goto）

4.2.3 比较操作

如果基本条件转移只在整数上操作，那么如何完成其他类型的比较？其他类型的比较是通过定义成返回整数的比较操作来完成的。例如，操作lcmp弹出两个长整数（像往常一样，两个长整数存成四个堆栈元素），并根据第一个元素是否大于、小于或等于第二个元素（就是堆栈顶部的元素）而压入1、0或者－1。例如，下面的代码片断就将存成长整数的局部变量#3与数1进行比较，当且仅当存储的值较大时就将控制转移到Somewhere：

```
lload_3          ; 装入局部变量#3（及#4）
lconst_1         ; 将1作为长整数压入，用于比较
lcmp             ; 比较大小，压入整数解
ifgt  Somewhere  ; 如果#3>1，转到Somewhere
; 如果运行到了这里，则#3 <=1
```

在堆栈按序操作中有非常重要的一点。指令序列lload_1、lload_3、lcmp将首先压入局部变量#1/#2，接着压入#3/#4，然后再做比较。比较操作对顺序是敏感的，如果是#1/#2在堆栈的顶部，就会给出一个不同的结果。要记住这个次序，只要这样想：从堆栈顶部第二个元素减去堆栈顶部元素。压入的结果就是这个差的符号(＋1、0或－1)。见图4-5的例子。

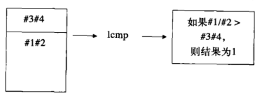


图4-5 lcmp指令示意图

比较浮点数和双精度数是类似的，但因为IEEE表示的复杂性，使这个问题有点棘手。具体说，就是IEEE 754允许用某种特殊的位模式来表示“不是一个数”，简写为NaN。这些特殊的模式在实质上意味着先前在某处的计算是完全错误的（如试图对复数求平方根或者用0除0）。通常与NaN进行比较没有意义，但程序员有时有一个特殊的解释。

例如，假设一个学院定义一个优等生列表，包括了所有平均分（grade point average，gpa）在3.50以上的学生。为了确定一个学生是否在这个优等生列表中，用Pascal编写的一个简短程序片断如下所示：

```
if (gpa > 3.50) then
      onhonorslist := true;
```

一个没有gpa的学生（例如，第一学期的学生或因病各科都未完成的学生）将不可能在优等生列表的考虑之内。换句话说，如果一个学生的gpa为NaN，将被视作小于3.50。然而，这个学生不应因gpa太低而被开除。NaN应该比适当的下限分数高。JVM和jasmin提供了两个不同的比较指令来定义这些情况。指令fcmpg比较堆栈顶部的两个浮点数，并根据结果将1、0或－1压入，例外情况是若两个数中有NaN，则结果是1。而若两个数中有NaN，fcmpl则返回－1。同样，比较两个双精度数的类似指令是dcmpg和dcmpl，这两个指令的行为也存在类似差异。

上面Pascal片断的jasmin等价程序大致如下：

```
fload_1       ; 从#1作为浮点数装入gpa
ldc 3.50      ; 装入gpa下限分数（3.50）
fcmpl         ; 将gpa与下限分数比较，NaN是“低”
ifle Skip     ; 转到Skip，gpa>=下限分数
iconst_1      ; 压入1（布尔值：true）
iconst_2      ; 将“true”放入到#2（作为整数/布尔数）
Skip:         ; 做需要对该学生做的其他事情
```

4.2.4 组合操作

如前所述，像短整数和布尔数这样的二类类型并未直接得到支持，在计算中必须作为整数类型看待。更奇怪的是，jasmin没有提供一种方式来计算比较两个整数的结果！替代方法是，比较是通过一些将比较操作与内置的条件转移相结合的快捷操作来完成的（如表4-2所示）。

表4-2 在jasmin中比较/转移指令的组合

if_icmpeq	如果第二个元素等于栈顶元素则转移
if_icmpne	如果第二个元素不等于栈顶元素则转移
if_icmplt	如果第二个元素小于栈顶元素则转移
if_icmpge	如果第二个元素大于或等于栈顶元素则转移
if_icmpgt	如果第二个元素大于栈顶元素转移
if_icmple	如果第二个元素小于或等于栈顶元素则转移

所有这些操作的原理差不多是一样的。CPU首先从栈中弹出两个int型元素，然后像icmp指令那样进行比较，而比较结果不是压入栈中，而是程序通过修改程序计数器立即跳到（或不跳）接近的位置。

4.3 建立控制结构

在高级语言中，控制结构的主要优势是相对来说容易理解。在（JVM或任何其他计算机上）进行汇编语言编程时尽量最大程度地保持这种易于理解性也是有益的。这样做的一种方法就是维护一个类似的代码逻辑块结构，像高级控制结构一样组织。

4.3.1 if语句

对4.2.3节关于在jasmin中如何表达高级控制的例子进行扩展，编写无错误和易理解代码的关键是保留一个类似的模块结构。传统的if/then语句有至多三个部分：一个布尔表达式，该表达式求值为真时要执行的一组语句，为假时要执行的另一组语句。这个C++或Java块是：

```
if (a > 5) {
        // (if模块)
        // 做一些事情
        // 需要几行
} else {
        // (else模块)
        // 做其他一些事情
        // 需要几行
}
        // 做需要做的其他任何操作
```

上述模块可用jasmin写成如下等价的模块（假设a是在#1中的一个长整数型）：

```
lload_1    ; 如果a在#1中
ldc_2 5    ; 装入5
lcmp       ; 比较a和5
ifle  Else
; 如果运行到了这里，则a > 5，因而执行的是if子句
; 在子句结束后，跳过else子句
; 要通过一个无条件goto跳过
; ----这对应于前面的if模块
goto  Quit
Else:
    ; 如果运行到了这里，则a <= 5，因而执行的是else子句
    ; ----这对应于前面的else模块
Quit:
    ; 做需要做的其他操作
    ; ----这对应于前面if语句后面的语句
```

这个代码和高级代码之间在模块结构上的相似性应该是显然的。特别是，在两个代码示例中都有相对于if模块和else模块的连续的指令，这些指令按序从头至尾执行。jasmin在这些模块之间插入代码并按此设置PC。在细节上，要注意这个例子中的测试有点不同，即在条件的反面成立时，不是直接转移到if子句，而是跳过if子句（到else子句）。

复杂的布尔条件可通过适当地使用iand、ior以及其他指令，或者通过反复的组合来解决。例如，我们可用如下程序来检验a是否处于5和10之间（a>5且a<10）的范围：

```
lload_1    ; 如果a在#1中
ldc_2 5    ; 装入5
lcmp       ; 比较a和5
ifle  Else
; 如果运行到了这里，则a>5
lload_1    ; 如果a在#1中
ldc_2 10   ; 装入10
lcmp       ; 比较a和10
ifge  Else
; 如果运行到了这里，则a>5且a<10
; 程序如前继续
```

4.3.2 循环

在很多编程语言中有两类基本的循环：程序在开始时测试条件的循环和测试在结尾测试条件的循环。第二种已举例说明。我们保留循环的内部模块结构，在模块的顶端置一个标号，然后在条件不能使循环退出时跳回到这个顶端。程序大致如下：

```
LoopTop:  ; 完成与循环相关的操作

          iload_1       ; 装入要比较的第一个操作数
          iconst_m1     ; 装入要比较的第二个操作数
          ifne LoopTop  ; (该循环持续进行直至#1=-1)
          ; 结束循环
```

这个程序实现的就是Pascal中的repeat循环，或者是C、C++或Java中的do/while循环。对于更传统的while循环，条件置于循环体的前面，循环体后跟一个无条件goto。如果循环退出条件满足，控制就被转移到循环外的首条语句，如下所示：

```
Control:
          iload_1       ; 装入比较的第一个操作数
          iconst_m1     ; 装入比较的第二个操作数
          ifeq LoopOut  ; (该循环持续进行直至#1=-1)
          ; 完成与循环相关的操作

          goto Control  ; 循环后，跳转并重新检测条件
LoopOut:  ; 你只能通过上面的条件转移运行到这里
```

这个代码片断执行的是与while (i != 1)等价的循环。还可以采取另外一种方式，就是保持do/while结构，但在底部通过一个无条件转移进入循环，如下所示：

```
          goto LoopEnd  ; 直接跳转到循环测试
LoopTop:  ; 完成与循环相关的操作

LoopEnd:
          iload_1       ; 装入比较的第一个操作数
          iconst_m1     ; 装入比较的第二个操作数
          ifne LoopTop  ; (该循环持续进行直至#1=-1)
          ; 结束循环
```

这就节省了一条goto语句的执行。

while和for循环的等价性已众所周知。一个由计数器控制的循环如下所示：

```
for (i=0;i<10;i++)
      //做一些事情
```

该循环等价于:

```
i = 0;
while (i<10) {
      //做一些事情
      i++;
}
```

这样也就能容易地用上面的框架写出。最重要的改变是需要用一个局部变量作为计数器，要不是这一点不同，while结构就几乎是完全重复的。

```
          bipush 0      ; 采用#1作为i，初始时设为零
          istore_1

          goto LoopEnd  ; 直接跳转到循环测试
LoopTop:
          ; 完成与循环相关的操作

          ; #1递增
          iload_1       ; 从#1中装入i
          iconst_1      ; 装入1，用于递增
          iadd          ; 我们现在已经完成了i++
          istore_1      ; ... 并将i存入到#1（再一次）

          ; 为提高效率，上面四行可用一个操作代替：iinc 1 1
LoopEnd:
          iload_1       ; 装入比较的第一个操作数（i）
          bipush 10     ; 装入比较的第二个操作数(10)
          ifle LoopTop  ; （该循环持续进行直至#1＝－1）
          ; 只有当#1 >= 10时结束循环
```

4.3.3 转移指令的细节

从程序员的角度看，采用goto语句和标号是相当简单的。在你希望控制跳转到的地方加一个标号，程序就自动地跳转到那里。在简单的外表下，其内部的实现却更复杂一些。计算机实际上存储的不是标号，而是偏移（offset）。例如，在字节码中，goto指令（操作代码0xA7）后接一个带符号的2字节（短）整数。这个整数就是用作为程序计数器的改变量。换句话说，该指令执行后，PC的值就变为PC＋offset。如果偏移值是负数，其效果就是将控制移回到以前运行过的语句（就像前面的重复循环的例子中所示），而如果偏移值是正数，程序就向前跳（就像if/then/else例子中所示）。在理论上，用值为0的偏移是完全可以的，但这意味着该语句没有改变PC，因此会产生无限循环。一个为0的偏移对应于下面的jasmin语句：

```
Self:
        goto Self
```

这也许不是程序员想要的。

那么，程序员是如何计算这些偏移的呢？幸运的是，他根本就不用计算。这是jasmin之类的汇编器的任务也是它的主要优势。程序员只要简单地使用就可以了，确定偏移应该是多少是汇编器的任务。对于这种实现还有一个潜在的更严重的问题（仍以程序员的角度看），就是只有2个字节的（有符号）偏移，不可能有比大约32 000字节更长的跳转（在前后方向上都是如此）。对于确实长的程序会出现什么情况呢？

这在jasmin中用两种方式解决。首先，除了goto指令以外，还提供一个goto_w指令（有

时称为“宽goto”，实现成为操作代码0xC8）。它在多个方面都类似于常规的goto，但后跟一个更长的整数，使得程序员能前跳或后跳多达20亿字节左右。由于单独的JVM方法不允许大于2^{16}（大约64 000）字节长，如果你写了一个更长的方法，或许就要将其分成若干个部分，这个新的操作代码解决了这个问题。当遇到像goto Somewhere的语句时，决定用什么操作代码仍然是汇编器的工作，它可用0xA7或者0xC8指令。如果需要的偏移过大，不能装到短整数中，就自动将程序员的goto转换成机器内部的goto_w。

一个更严重的问题是不存在对应ifne_w的直接对等指令或者其他宽的条件转移指令。然而，汇编器仍能在无需程序员的知识或合作的情况下设计出等价功能。一个转移到遥远位置的宽条件转移的功能可用一个“绕过转移的转移”的方法来模拟，如图4-6所示。

```
        ; 真正需要的是“ifne DistantLocation”
        ; 但由于太远，不能在单步达到
        ; 不幸的是，不存在ifne_w
        ; 汇编器会实现这个功能

        ifeq Skip                ; 注意测试的是相反情况
        goto_w  DistantLocation  ; 因为我们在绕过转移
Skip:   ; 做其他一些事情
```

图4-6　一个“绕过转移的转移”的例子，即有条件地跳转到遥远的位置

就像计算转移大小一样，一个好的汇编器对于这种情况总有正确的做法。

使用转移语句的最严重的问题是它们没有以任何方式为程序员提供局部模块结构。很多程序员依赖于能在模块内重新定义局部变量的便利，如下所示：

```
if (x > 0) {
    int x; // 这是一个新的x，与以前的无关
    x = -1;
        ...
}
//      这里，旧的x重新出现并重新请求其旧值
```

在汇编语言中没有这种便利性。所有的局部变量（以及堆栈）在跳转之前、期间、之后都保留其值。如果在跳转前，一个重要的变量存储在#2，而下一条语句是fstore_2，则该重要变量将被重写。模块结构对于如何考虑汇编语言程序是一个重要的便利，但它没有提供像信息隐藏等任何实际的防范措施以避免错误和误用。

4.4 示例：Syracuse数

4.4.1 问题定义

作为一个将这些内容结合到一起的例子，我们将探讨Syracuse数（即$3N+1$）猜想。这个问题可追溯到古典数学，一个古希腊人注意到一些简单的算术规则会产生惊人的、不可预测的行为，这些规则是：

- 如果数N是1，则停止。
- 如果数N是奇数，则将N变为$3N+1$，并继续。
- 如果数N是偶数，则将N变为$N/2$，并继续。

人们在早期发现，若你从任何一个正整数开始进行，这个过程看起来总会停止（到达1），但没有人能实际地证明这个猜想。进一步地说，没有人能发现一个一般的规则以预测需要多

少步才能到达1。有时这个过程恰巧非常快：

16→8→ 4→2→1

但相近的数字可能需要不同的次数：

15→ 46→23→70→35→106→ 53→160→80→
40→20→10→5→16→8→4→2→ 1

有时需要非常大的次数。甚至在目前，也没有人知道需要多少步，甚至是否总是到达1（虽然数学家采用计算机已经发现所有小于几十亿的数都会收敛到1）。你自己试着做一下：从81开始，你认为需要多少步才能到达1？

4.4.2 设计

更好的办法是不用你自己去做，而是让计算机做这件事情。图4-7给出了针对开始于81直至到达1的算法的伪代码。用jasmin实现这个算法就会快速得出这个猜想的答案。

```
 (1)     count_of_steps <- 0
 (2)     current_value <- 81
 (3)     while (current_value != 1)
 (4)           if (current_value is odd)
 (5)                 current_value <- (current_value * 3) + 1
 (6)           else
 (7)                 current_value <- (current_value / 2)
 (8)           endif
 (9)           count_of_steps <- count_of_steps + 1
(10)    endwhile
(11)    final answer is count_of_steps
```

图4-7 用伪代码测试Syracuse猜想

图4-7中的代码需要两个整数变量，为计算简单，我们将它们看作整数类型（而不是长整数）。特别是，count_of_steps可存储为局部变量#1，而current_value可被存储为局部变量#2。步骤1、2、5、7、9的算术运算可用第二章的技术来执行。第11行的输出可用若干种方式完成。这里我们选择较简单的一种，就是只打印最终结果。

从第4到第8行用到的if/else结构可用第4.3.1节的代码块来建模。特别地，我们可用除以2取余数（irem）的方法来确定当前值是否是奇数。如果结果等于0（if_icmpeq），则该数是偶数。图4-8中的代码显示出这一点。作为典型的结构化程序设计，从整体上看，这个模块在起始语句有单个入口，在底部有单个出口（标记为Exit:的那一点）。

```
        ; if/else的入口点
        iload_2        ; 装入current_value
        iconst_2       ; 还有2，以确定奇偶性
        irem           ; 计算余数
        iconst_0       ; 比较余数与0
        if_icmpgt CaseOdd，如果是奇数，就转到CaseOdd

    CaseEven:          ; 从技术上说，我们不需要这个标号
                       ; 因为永远不会转移到这个位置

        ; 将#2除以2并重新存储
        iload_2        ; 装入当前值
        iconst_2       ; 压入2用于除
        idiv           ; 做除法
        istore_2       ; 并压入新值
```

图4-8 Syracuse数计算代码内部模块的if/else结构

```
    goto        Exit     ; 跳过CaseOdd模块

CaseOdd:
    ; 将#2乘以3并加1
    iload_2          ; 装入当前值
    iconst_3         ; 压入3用于乘
    imul             ; 做乘法（现在存储的值就是3*N)
    iconst_1         ; 压入1用于加
    iadd             ; 做加法（存储的值就是3*N+1)
    istore_2         ; 并将新值存起来

Exit:
    ; if/else分支的共同出口点
```

图4-8 （续）

全部的代码块将用于while-loop结构的内部，如图4-9所示。

```
; while的入口
LoopEntry:
    iload_2  ; 从#2中装入current_value
    iconst_1; 将current_value与1比较
    if_icmpeq   LoopExit ; 如果相等就转移到LoopExit

; 做奇偶计算的必要语句

; 递增count_of_steps的必要语句

        goto  LoopEntry ; 无条件转移到循环的顶部
                        ; 并重新检测

LoopExit:
; while循环的出口点
```

图4-9 Syracuse数计算代码主循环的while结构

4.4.3 解答与实现

这里给出了完整的解答。

补充资料

```
; 样板
.method public <init>()V
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 2         ; 这里没有复杂的计算
    .limit locals 3        ; #0是保留的（通常情况）
                           ; #1是循环计数器
                           ; #2是N的值

    iconst_0               #1 <- 0
    istore_1

    bipush 81              ; #2 <- 81
    istore_2
```

```
LoopEntry:
    iload_2              ; 从#2装入current_value
    iconst_1             ; 将current_value与1比较
    if_icmpeq  LoopExit  ; 如果相等，就转移到LoopExit


    ; 做一些奇偶计算所必要的语句
; if/else的入口点
    iload_2              ; 装入current_value
    iconst_2             ; 还有2，以确定奇偶性

    irem                 ; 计算余数
    iconst_0             ; 将余数与0比较
    if_icmpgt CaseOdd    ; 如果是奇数，则转移到CaseOdd

    ; 只有#2是偶数时我们才能到这里
    ; 将#2除以2并重新存储
    iload_2              ; 装入当前值
    iconst_2             ; 压入2用于除
    idiv                 ; 做除法
    istore_2             ; 存储新值

    goto Exit            ; 跳过CaseOdd模块
CaseOdd:
    ; 将#2乘以3再加1
    iload_2              ; 装入当前值
    iconst_3             ; 压入3用于乘
    imul                 ; 做乘法（现在存储的值是3*N）
    iconst_1             ; 压入1用于加
    iadd                 ; 做加法（存储的值是3*N+1）
    istore_2             ; 并存储新值

Exit: ; 执行递增count_of_steps的必要语句
    iinc 1 1             ; 递增循环索引

    goto LoopEntry       ; 无条件转移到顶部，并重新检测


LoopExit:
    ; 打印结果到System.out（通常如此）
    getstatic java/lang/System/out Ljava/io/PrintStream;
    iload 1              ; 装入循环计数器用于打印
    invokevirtual java/io/PrintStream/println(I)V

    return               ; 完成
.end method
```

4.5 表跳转

多数高级语言还支持多路判定的概念，这在Java中表示成一个switch语句。作为这种多路判定的使用示例，请看一个试图计算一个月中有多少天的程序，如图4-10所示。

显然，任何多路转移都能以一组两路转移（如if/else语句）来看待，并据此编写程序。JVM还提供了一个快捷方式（事实上是两个快捷方式），使得在某些条件下代码执行得更简单更快速。主要的条件就是：分支（case）标号（例如，就是在前面例子中的数字1到12）必须是整数。

如前所述，没有直接的模块结构的概念。机器提供的是一个多路转移，计算机根据堆栈顶部的值，转移到这若干个目的地之中的一个。lookupswitch指令的一般格式包含一组的值：标号对。如果值匹配堆栈顶部元素，则控制就传递到该标号。

```
switch (monthno) {
     // 九月、四月、六月、及11月有30天
     case 9 :
     case 4 :
     case 6 :
     case 11 : days = 30; break;
     //  ...所有其他月份都有31天
     case 1 : case 3 : case 5 : case 7 : case 8 : case 10 : case 12:
              days = 31; break;
     // 唯独除了二月以外（不考虑闰年）
     case 2 : days = 28; break;
     default : System.out.println("Error : Bad month!");
} // switch结束
```

图4-10 在Java和C语言中的多路转移语句

```
        iload_1              ; 假设"monthno"存储在#1中
        lookupswitch         ; 多路转移开始
            1    : Days31    ;
            2    : Days28    ;
            3    : Days31    ;
            4    : Days30    ;
            5    : Days31    ;
            6    : Days30    ;
            7    : Days31    ;
            8    : Days31    ;
            9    : Days30    ;
           10    : Days31    ;
           11    : Days30    ;
           12    : Days31    ;
         default : ERROR    ;
Days28:
         bipush 28           ;装入28天
         istore_2            ;并分配到天（#2）
         goto ExampleEnd
Days30:
         bipush 30           ;装入30天
         istore_2            ;并分配到天（#2）
         goto ExampleEnd
Days31:
         bipush 31           ;装入31天
         istore_2            ;并分配到天（#2）
         goto ExampleEnd
Error:
         ;错误发生时完成的动作，如getstatic .../System/out
         goto ExampleEnd
ExampleEnd:
         ;做必要的事情
```

以上开始于lookupswitch并结束于default的大约12行语句实际上是一个非常复杂的机器指令。与其他已经讨论过的语句不同，这条指令所接受参数的数量是变化的，因而jasmin汇编器（以及JVM字节码解释器）的任务相应地也就复杂。在JVM机器码中（与在Java中不同），default这个分支是必须要有的。像其他转移语句一样，存储的值是相对于当前程序计数器的偏移。与多数其他的语句不同（除了goto_w），偏移被存储成4字节的量，使得在方法内能跳转到“遥远”的位置。

在上面例子中，如果值不仅是整数，而且是连续的整数，意思是从初始值（1，January）

运行直至最终值（12，December）而没有中断或跳过，这样的话，JVM就提供了另一个用于多路判定的快捷操作。该思想很简单，就是如果最低的可能值是36，则下一个值就是37，然后是38，依此类推。如果定义了最低值和最高值，其余的值就可像表一样填充。因此，这个操作称为tableswitch，使用方法如下：

```
        iload_1                 ; 假设"monthno"存储在#1中
        tableswitch 1 12        ; 开始多路转移，从1到12
            Days31   ;
            Days28   ;
            Days31   ;
            Days30   ;
            Days31   ;
            Days30   ;
            Days31   ;
            Days31   ;
            Days30   ;
            Days31   ;
            Days30   ;
            Days31   ;
            default : Error    ;
Days28:
         bipush 28              ; 装入28天
         istore_2               ; 并分配到天（#2）
         goto ExampleEnd
Days30:
         bipush 30              ; 装入30天
         istore_2               ; 并分配到天（#2）
         goto ExampleEnd
Days31:
         bipush 31              ; 装入31天
         istore_2               ; 并分配到天（#2）
         goto ExampleEnd
Error:
         ; 错误发生时完成的动作，如getstatic .../System/out
         ; 并打印出一个错误信息
         goto ExampleEnd
ExampleEnd:
         ; 做必要的事情
```

补充资料

lookupswitch/tableswitch的机器码

lookupswitch和tableswitch都涉及可变数量的参数，因而在字节码中就有复杂的实现。实质上，存在一个关于有多少个参数的“隐藏的”隐含参数，使得计算机知道下一个参数从哪里开始。

对于lookupswitch的情况，jasmin汇编器将为你统计值：标号对的数量。所生成的字节码不仅包含lookupswitch操作代码（0xAB），还有一个4字节用来对非default分支进行计数。每个分支存储成4字节整数（值），并且当该整数匹配堆栈顶部元素时就取得相应的4字节偏移。

对于tableswitch的情况，值可通过开始和最终值计算出来。一个tableswitch语句在内部存储成操作代码字节（0xAA）、低值和高值（存储成4字节整数），最后是一组连续的4字节偏移（对应于值*low*、*low*+1、*low*+2，…*high*）。两个指令更详细介绍见附录B。

在tableswitch例子中唯一不同的操作是tableswitch本身。其余的代码是相同的。重要的不同点与表的结构相关。程序员需要定义变量可取的最低值和最高值，然后对标号从小到

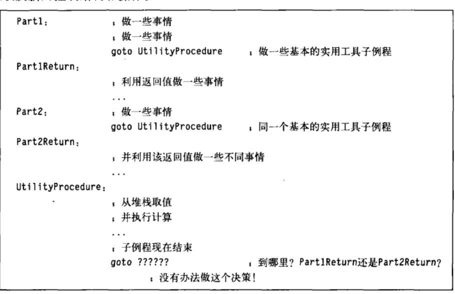
大排序，而不是明确地写出所有的标号：值对。这些都是由表结构自动地完成的。在上面的例子中，第四个标号（Day30）是自动地附于第四个值的。如前所述，default分支是必须要有的。

当然，这些跳转表是否会对程序的效率有所帮助，要看具体情况和具体问题。一个switch语句总是能写成一组适当的有合适复杂条件的if语句。在某些情况下，计算一个布尔条件比枚举各种情况可能要容易一些。

4.6 子例程

4.6.1 基本指令

基于分支控制的一个主要局限性是在一个代码块被执行后，它将自动地将控制转移回到一个不可改变的点。与高级编程语言中的传统过程不同，要建立一个代码块使得从程序中任何一点运行然后再回到原来的地方，这是不可能的。（见图4-11）。为了做到这一点，需要更多的信息以及新的控制结构和操作。



```
Part1:
                    ; 做一些事情
                    ; 做一些事情
                    goto UtilityProcedure      ; 做一些基本的实用工具子例程
Part1Return:
                    ; 利用返回值做一些事情
                    ...
Part2:
                    ; 做一些事情
                    goto UtilityProcedure      ; 同一个基本的实用工具子例程
Part2Return:
                    ; 并利用该返回值做一些不同事情
                    ...
UtilityProcedure:
                    ; 从堆栈取值
                    ; 并执行计算
                    ...
                    ; 子例程现在结束
                    goto ??????                ; 到哪里? Part1Return还是Part2Return?
                          ; 没有办法做这个决策!
```

图4-11　子例程返回问题

需要的主要信息当然是控制来自何处，因为该位置要作为返回点。这需要两个基本的修改：首先，有一个也存储（到某处）跳转前PC值的转移指令；其次，有另一种会返回到一个可变位置的转移指令。采用这种语义写成的代码块通常叫做子例程（subroutine）。

JVM提供一个jsr（跳转到子例程）指令。在技术上，为满足这个需要，它提供了两条指令，jsr和jsr_w，这类似于goto和goto_w。当jsr指令执行时，控制被立即传递（就像用goto指令一样）到标号，标号的偏移存储在字节码中。在此发生之前，计算机计算出值（PC+3），这是紧接着jsr指令本身的下一条指令的地址。该值作为一个常规的32位（4字节）量被压入到堆栈，就在此时控制被传递到标号。

补充资料

jsr的机器语言

为了确切地理解指令如何工作，让我们详细地考察jsr指令的机器码。jsr助记符对应于单字节（0xA8）。jsr后接一个2字节的偏移，存储成一个带符号的短整数。假设存储器位置0x1000－0x1003存储了如表4.3的模式。

表4-3 jsr指令的字节码结构

位置	0x1000	0x1001	0x1002	0x1003
字节值	0xA8	0x00	0x10	0x3b
解释	jsr	Integer：0x0010＝16	istore_0	

当jsr指令执行时，PC将有值0x1000（根据定义）。存储器中的下一条指令（istore_0）存储在位置0x1003。

所有提供子例程调用的机器也提供从子例程返回的指令。如何在基于堆栈的计算机上完成这一点是相当容易理解的。返回指令将会检查堆栈顶部并使用存储在那里的值，这个值是由jsr指令压入作为返回的地址。这个位置就成为一个隐含分支的目标地址，控制返回到主程序，计算继续正常运行。

在JVM上事情就稍微复杂一些，主要是安全性的原因。ret指令不检查堆栈以获得返回位置，而是接受局部变量数作为一个参数。任何子例程必须执行的第一个任务就是将jsr指令（隐含地）压入的值存储到一个适当的局部变量中。做完这件事之后，就不再改变这个位置。试图对存储地址执行计算在最好的情况下也是危险的，通常是会引起误导，而在Java和相关的语言中则是完全非法的。这也是安全性模型和验证器通常要避免的事情。

4.6.2 子例程示例

为什么要用子例程

子例程的一种常用用法与Java方法或C++过程很类似：就是执行一个特定的固定任务，而这个任务在程序中的若干个位置都可能需要。一个明显的例子就是打印某些东西。正如我们在前面的一些例子中所看到的，诸如打印一个串等事情可能相当棘手，需要用几行来完成。为了使程序更容易和更高效，一个熟练的程序员可将这些行做成一个子例程模块，并用jsr和ret访问。

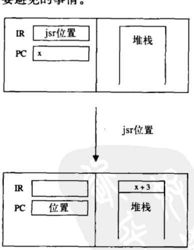


图4-12 jsr指令和堆栈

*子例程*Max（int A int B）

让我们从一个算术运算子例程的简单例子开始。这个简单例子计算（并返回）两个整数之中的较大者[⊖]。让我们假设两个数（作为整数）在堆栈上，如图4-13所示。

⊖ 当jsr指令执行时，值0x1003（作为地址）被压入到堆栈，PC的值改变为0x1000+0x0010，即0x1010。这意味着程序将向前跳转16个字节并开始执行一段新的代码。当ret指令执行时，将返回位置0x1003的iconst_0指令。jsr_w指令类似，只不过它涉及4字节的偏移（因而可进行更长的跳转），并将值PC+5压入。（见图4-12）



图4-13 max(A,B)的堆栈结构

if_icmp??指令将为我们做适当的比较，但会弹出（并销毁）这两个数。所以在做这件事之前，我们应该采用dup2复制堆栈顶部的两个字。在做之前，所有子例程必须处理返回地址。为简单起见，我们将其存储到局部变量#1。

```
      ; 假设这通过jsr Max来调用
      ; 两个整数已经在堆栈上
Max:
      astore_1              ; 将返回地址存储到#1

      dup2                  ; 拷贝两个参数以备后用
      if_icmpgt Second      ; 如果A<B，则转移
First:
      ; A >= B，所以需要从堆栈中删除B
      pop
      goto Exit
Second:
      ; A < B，所以需要将A交换到顶部并删除它
      swap
      pop
Exit:

      ret 1     ; 返回到存储在#1的位置
```

*子例程*PrintString(String s)

作为第二个例子，我们将写（并使用）一个子例程，功能是输出一个固定的串，该串被压在堆栈上。让我们首先写出子例程：

```
      ; 假设这通过jsr PrintDream调用
      ; 要打印的串已经被压入到堆栈
      ; 位于返回地址下面
PrintString:
      astore_1  ; 将返回地址存储到#1

      ; 假设要打印的串已经在堆栈上

      ; 你应该已经了解下面三行
      getstatic java/lang/System/out Ljava/io/PrintStream
      swap       ; 将参数按正确的顺序放置
      invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

      ret 1     ; 返回到存储在#1中的位置
```

新的指令astroe_1是?store_N的又一个例子，但它存储的是一个地址（类型‘a’）而不是一个浮点数、双精度数、整数或者长整数。getstatic和invokevirtual这两行你应该已经熟悉了。该代码假定调用环境在执行跳转到子例程的调用之前已将串压入，这样你只需要压入System.out对象和调用必要的方法。一旦做完，ret指令就使用新近存储到#1的值作为返回地址。然而，要注意仍需要做点“信息隐藏”或模块结构的工作，因为这个子例程会不可挽回地摧毁已存储在局部变量#1中的任何信息。如果程序员想要采用这个代码模块，他就需要注意这个行为。更一般地，对于任何子例程，重要的是要知道哪些局部变量是子例程用到的和不用的，因为它们与主程序所使用的是同一组局部变量。

使用子例程

主程序可写得任意复杂，并可能涉及对该子例程的多次调用。打印几句诗怎么样？

```
; 压入要打印的第一个串/行
ldc "'Twas brillig, and the slithy toves"
; 调用在PrintString开始的子例程
jsr PrintString    ; 这行以后马上返回
                   ; 注意不需要标号

; 以类似的方式继续
ldc "Did gyre and gimble in the wabe."
jsr PrintString

ldc "All mimsy were the borogroves"
jsr PrintString

ldc "And the mome raths outgrabe."
jsr PrintString

; 永远不要忘记引用诗的出处
ldc "(from Jabberwocky, by Lewis Carroll)"

return    ; 退出方法，因为我们已经打印了这首诗
```

这个程序的一个完成版本在图4-14中给出。（实际上，在图中有一个故意的错误。有一行不会打印出来。你能发现是哪一行和为什么吗？更重要的是，你知道如何改正这个错误吗？）

```
; 打印刘易斯·卡洛尔的诗Jabberwocky的第一节的程序

. class public jabberwocky
.super java/lang/Object

; 通常的样板
.method public <init>()V
    aload_0

    invokespecial java/lang/Object/<init>()V
    return
.end method

.method public static main(Ljava/lang/String;)V
    .limit stack 2
    .limit locals 2

; 压入要打印的第一个串/行
ldc "Twas brillig, and the slithy toves"
; 调用在PrintString开始的子例程
jsr PrintString    ; 这行以后马上返回
                   ; 注意不需要标号

; 以类似的方式继续
ldc "Did gyre and gimble in the wabe."
jsr PrintString
```

图4-14 子例程示例的完整程序（包含一个错误）

```
ldc "All mimsy were the borogroves"
jsr PrintString

ldc "And the mome raths outgrabe."
Jsr PrintString

; 永远不要忘记引用诗的出处
ldc "(from Jabberwocky, by Lewis Carroll)"

return    ; 退出方法，因为我们已经打印了这首诗

    ; 假设这个子例程通过jsr PrintString调用
    ; 要打印的串已经压入到堆栈
    ; （在返回地址的下面）

PrintString:
astore_1    ; 将返回地址存储到#1

; 假设要打印的串已经在堆栈上

; 你应该已经了解下面三行
getstatic java/lang/System/out Ljava/io/PrintStream;
swap    ; 将参数按正确的顺序放置
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

ret 1    ; 返回到存储在#1中的位置

.end method
```

图4-14 （续）

4.7 例子：π的蒙特卡洛估计

4.7.1 问题定义

现在每个人都知道π的值（3.14159以及更多几位）。数学家是如何计算出它的值的呢？在几个世纪中已经试验了很多方法，包括一种不是因其数学上的复杂性而是因为其简单性而闻名的。我们在这里给出这种方法的一个变型，该方法首先由George de Buffon使用。

考虑（随机地和均匀地）向图4-15所显示的图形中掷飞镖。我们假设飞镖可能击中正方形区域内的任何地方。特别是，一些飞镖会击中圆内而另一些没有击中圆内。由于正方形的边长是两个单位，故其总面积是4。圆的面积是πr^2，计算得出面积为π。这样，我们预计飞镖击中圆内的比率将会是π/4。

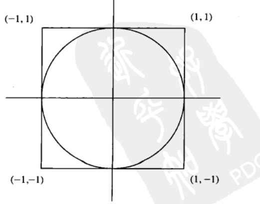


图4-15 蒙特卡洛π估计的图示

换句话说，如果我们向图中投掷10000个飞镖，我们预计会有大约7854个会击中圆内部。我们甚至可将注意力关注在右上象限，并得出同样的结果。

这种探讨方法通常称为蒙特卡洛模拟（Monte Carlo simulation）。当你对概率空间的精确参数不确信时，这可成为探索大概率空间的非常有力的方法。它也属于那种计算机能显示其优越性的任务，因为简单的计算（飞镖击中哪里？击中的是圆内还是圆外）可被重复成千上百万次直到你得到一个足够精确的解。

4.7.2 设计

3.4.1节讨论了随机数生成的一点理论和实际知识。在那一节中开发的代码可为我们提供随机的整数，但这里要做一些变化。主要的变化是（在单位圆中的）每个位置由两个数来定义，因此就需要在两个不同的地方调用生成器。这就意味着要使用子例程。

除此以外，该程序需要两个计数器，一个用于投掷的飞镖数，一个用于击中圆内的飞镖数。对于任意一个飞镖，如果其击中的位置是 (x, y)，则其击中圆内当且仅当$x^2 + y^2 \leqslant 1$[⊖]。

解决这个问题的伪代码大致如图4-16所示。问题本身的结构因涉及重复的随机点生成以及统计成功和失败次数，所以也表现出某种循环的特质。关系到成功和失败（在单位圆的内部或外部）的实际决策将用一个if/then等价结构来实现。总之，这个程序可采用同样适用于其他编程语言（像Java或Pascal）的高级结构来着手设计。

```
(1)     total_hits <- 0
(2)     for (total_darts := 1 up to 10000)
(3)          generate (x,y) position for new dart
(4)          if ( (x,y) inside circle )
(5)               total_hits <- total_hits + 1
(6)          endif
(7)     endfor
(8)     final answer is (total_hits / 10000)
(9)     FINAL final answer is (total_hits / 10000) * 4
```

图4-16 用蒙特卡洛方法计算π的伪代码

在这个模块执行足够时间后，成功次数与总执行次数的比率应该接近π/4。

对于变量，我们将需要至少两个整数用作计数器，另一个位置用来保存随机数种子的当前值，还有两个位置用来保存当前飞镖的 (x, y) 坐标，还有第6个变量用来保存从随机数生成子例程返回的位置。必要的控制结构已经都在前面部分讨论过了。

特别是，如果变量#4和#5分别保存（作为浮点数）了x和y坐标，则只要飞镖在圆内，下面的代码模块就会递增在#1中的计数器。

```
          fload_4          ; 从#4装入x坐标
          dup              ; 对其做平方
          fmul
          fload_5          ; 从#5装入y坐标
          dup              ; 对其做平方
          fmul
          fadd             ; 计算x^2 + y^2
          fconst_1         ; 压入1用于比较
          fcmpg            ; 做比较
          ifgt Skip        ; 如果在圆外，则转到Skip
          iinc 1 1         ; 递增在#1中的计数器
Skip:     ; 做需要的事情
```

⊖ 如果你不确信这个公式为什么可行，可以使用距离公式。

我们可以修改3.4.2节的第一个随机数生成器，就能相当容易地生成浮点数，见下面的代码片断：

```
; 计算a*oldvalue
ldc2_w   65537         ; a是2^16+1，存储为长整数
iload_3                ; oldvalue被存储为局部变量#3
i2l                    ; 将oldvalue转换成为长整数
lmul                   ; 并做乘法

; add c
ldc2_w 5               ; c是5，存储为一个长整数
ladd                   ; 并加到a*oldvalue

; 获得mod m的余数
ldc2_w 4294967291 ; 为m装入值
lrem                   ; 计算模
l2i                    ; 转换回整数
dup                    ; 复制，为了存储
istore_3               ; 为下次存储新值

i2f                    ; 转换成浮点数
ldc 4294967291.0       ; 作为浮点数装入m
fdiv                   ; 做除法，得到[0,1)中的数

; 浮点数留在堆栈顶部
```

这个片断又会成为生成x和y坐标的子例程的核心。

4.7.3 解答与实现

这里给出了完整的解答。

```
; 通过蒙特卡洛模拟计算pi的程序

.class public pi
.super java/lang/Object

; 样板
.method public <init>()V
      aload_0
      invokespecial java/lang/Object/<init>()V
      return
.end method

.method public static main([Ljava/lang/String;)V
      .limit stack 4
      .limit locals 7

      iconst_0          ; 迄今投掷的飞镖数是0
      istore_1          ; 飞镖数在#1中

      iconst_0          ; 迄今投掷到内部的飞镖数是0
      istore_2          ; 投掷到内部的飞镖数在#2中

      iconst_1          ; 将随机数生成器的种子设为1
      istore_3          ; 随机数生成器种子在#3中

Head:
      iload_1           ; 装入投掷飞镖的数量
      ldc 10000         ; 与10000飞镖比较
      if_icmpgt End     ; 如果多于10000，退出循环
```

```
        jsr Random          ; 为x坐标获得随机浮点数
        fstore 4            ; 并存储到#4

        jsr Random          ; 为y坐标获得随机浮点数
        fstore 5            ; 并存储到#5

        fload 4             ; 从#4装入x坐标
        dup                 ; 求平方
        fmul
        fload 5             ; 从#5装入y坐标
        dup                 ; 求平方
        fmul
        fadd                ; 计算x^2 + y^2
        fconst_1            ; 压入1用于比较
        fcmpg               ; 做比较
        ifgt Skip           ; 如果在圆外，则转到Skip
        iinc 2 1            ; 递增在#2中的圆内飞镖数

Skip:
        iinc 1 1            ; 递增在#1中的投掷总飞镖数
        goto Head           ; 并转到循环的顶部

Random:
        astore 6            ; 存储返回值

        ; 计算a*oldvalue
        ldc2_w    65537     ; a是2^16+1，存储为长整数
        iload_3             ; oldvalue存储为局部变量#3
        i2l                 ; 将oldvalue转换为长整数
        lmul                ; 并做乘法

        ; add c
        ldc2_w 5            ; c为5，存储为长整数
        ladd                ; 并加到a*oldvalue

        ; 并获得mod m的余数
        ldc2_w 2147483647 ; 为m装入值
        lrem                ; 计算模
        l2i                 ; 转换回整数
        dup                 ; 复制，为了存储
        istore_3            ; 为下次存储新值

        i2f                 ; 转换成浮点数
        ldc 2147483647.0 作为浮点数装入m
        fdiv                ; 做除法，得到[0,1)中的数
        ret 6               ; 返回到存储于#6中的调用环境

End:
        iload_2             ; 装入内部的总飞镖数
        i2f                 ; 计算浮点数比率
        iload_1             ; 装入总飞镖数
        i2f                 ; 计算浮点数比率
        fdiv                ; 做除法，计算比率 (pi/4)
        ldc 4.0             ; 乘以4
        fmul                ; 得到最终的解

        ; 并打印
```



```
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap            ; 使参数顺序正确
        invokevirtual java/io/PrintStream/println(F)V
        return
.end method
```

4.8 本章回顾

- 当前正在执行的指令的位置存储在CPU内部的程序计数器（PC）中。在正常情况下，每次一条指令被执行，PC就递增以指向紧接着的指令。
- 某些指令可改变PC的内容，因而导致计算机改变其执行路径。这些语句通常称为转移语句或goto语句。
- 任何语句若是转移的目标则必须有一个标号（label）。标号只是一个字，通常开始于一个大写字母，标记其在代码中的位置。
- goto语句执行的是无条件转移。控制直接转移到其参数标记的点。
- if??系列的条件转移将控制转移到目标位置（也可能不转移）。这些指令从堆栈弹出一个整数，并根据该整数的大小和正负来决定是转移还是继续正常的取指-执行周期。
- ?cmp系列的语句用于对非整数类型做比较。这些指令从堆栈弹出两个适当类型的数，并根据第一个参数是否大于、等于或小于第二个参数而压入值为1、0或-1的整数。这个整数接着就可用于后续的if??指令。
- if_icmp??指令将icmp语句的功能与if??条件转移系列相结合成为一条指令。
- 事实上，像if/else语句和while循环这样的高阶控制结构必须在汇编语言级别用条件和无条件转移来实现。
- lookupswitch和tableswitch语句提供一种执行多路转移语句的方法，对于实现case或switch语句来说，这可能比一系列的if/else语句更高效。
- 子例程的工作原理是将PC的当前值压到堆栈，并在子例程结束时返回到先前存储的位置。在jasmin中，这分别对应于jsr和ret指令。与多数其他机器不同，为安全原因，ret指令期望在局部变量中寻找返回地址。因此，子例程中的第一个操作通常是astore操作，用以将（压入的）返回位置从堆栈存储到局部变量。

4.9 习题

1. 为什么JVM不允许直接将一个整数装入到PC？
2. 结构化编程的概念如何在汇编语言中实现？
3. “如果大于0或小于0则转移”指令在JVM指令集可得到吗？如果没有，你如何实现它？
4. NaN（不是一个数）大于或小于0吗？
5. 你如何在jasmin中建立一个if/else-if/else-if/else控制结构？
6. goto和goto_w之间有什么差别？有相应的ifne_w指令吗？
7. 在图4-8中的代码使用irem来确定一个数的奇偶性。Juola的“multiple cat skinning”定律指出做任何事情总有更多的方法。你能找出一种方法，用iand来确定一个数的奇偶性吗？用ior怎么做？
8. 用移位操作怎么做呢？
9. 图4-14中的错误是什么？解决方案是什么？
10. （本题针对高级程序员）本章中提出的jsr和ret的语义支持递归吗？并给予解释。

4.10 编程习题

1. 写一个程序，确定小于或等于N的2的最大幂。
2. 至少有两个不同的方法能针对前面的问题编写程序，一个使用移位指令，一个使用乘法/除法指令。在你的机器上哪个运行得更快？
3. 写一个程序，确定能装入一个整数变量的N的最大幂。再确定能装入一个长整数局部变量的N的最大幂。你如何判定什么时候发生溢出？
4. 写一个程序，使用密钥$N = 13*17$和$e = 11$实现RSA加密和解密（你可能要在Web上查找这个算法）。
5. a. 写一个程序，测试随机数生成器的优劣。特别地，该程序应该以大致相同的频率产生所有输出。随机生成从1到10的100 000个数。最小频率的数应该至少在最高频率的数的90%频率内。你的生成器优劣如何？
   b. 寻找产生一个好的生成器的a、c、m的值。
   c. 寻找产生一个坏的生成器的a、c、m的值。
   d. （本题针对高级程序员）对生成器的输出运行chi-平方测试以确定其优劣程度。





第二部分　真实计算机

第5章　通用体系结构问题：实际计算机

5.1　虚拟机的限制

JVM作为虚拟机已经在大多数真实计算机上得以实现。在某种意义上，JVM设计者已经取得了很大的成功——因为JVM是非常简单并且易于理解的体系结构，也是计算机组织与体系结构教学的最佳用机之一。然而，这种简明在某种程度上忽略了实际计算机器件的某些限制。例如，JVM中的每个方法（method）假定运行在其自包含的环境中；改变一个方法中的局部变量不会影响其他的方法。相比之下，只有一个CPU和一个主存储器的实际计算机意味着同时运行两个功能会引起对寄存器、存储器等的争用。你可以想象，如果一个支票填写程序开始接收数据（比如从一个游戏），同时开始打印支付星云学会剩余光子鱼雷的支票将会导致的混乱。

类似地，机器容量问题对于JVM或者类似的虚拟机来说却不是问题；对于所有实际用途，JVM虚拟机的栈具有无限的深度，能够容纳无限量的局部变量。对比之下，PowerPC（用于游戏机的芯片，近来也用于Macintosh）只有32个寄存器用于计算，而基于Pentium的Windows PC用于计算的寄存器就更少了。

JVM能够忽略的另外一个主要问题便是速度。要快速地执行一个JVM程序，你仅需要在一个快速的物理处理器上运行一个JVM的拷贝。相比来说，要建造一个快速的物理处理器，就需要解决具有难度和竞争性的工程问题。Intel和AMD的工程师一直在进行前沿探索，以便制造更快速的处理器。当然，世界上那些训练有素的工程师们正在逐步解决这一难题，虽然如何解决这些问题的细节已经超出了本书的范围，但是接下来的各节将会探讨一些提高性能的部件优化方法。

5.2　CPU优化

5.2.1　建造一个更好的捕鼠夹

改善计算机性能的最明显方式就是提高整体性能参数——例如，将计算机的字长从16位增加到32位。两个32位数的加法在一个32位的计算机上只需要一个加操作就可以完成，但是在一个16位的计算机上却要通过至少两次相加才能完成。类似地，将时钟速度从500MHz增加到1GHz能将每个操作减少到原来的一半时间，相当于机器的性能增加了一倍。

实际上，很少能像人们认为的那样有效。目前几乎所有的机器都是32位，32位寄存器对于大部分应用已经足够精确了。采用64位的寄存器能使程序员设计支持10^{24}数量级的快速数据操作，但是多久你需要用到一次这种数量级的操作呢？除此之外，如果存储器和总线的速度都没有CPU的速度快，即使CPU很快也不一定能带来多大的益处。

更为严峻的是，以这种提高速度的方式提高性能是昂贵的、困难的。例如，处理器算术运算器件的速度受限于晶体管的物理和电气响应特征，运行太快容易使其毁坏。即使能够制造速度很快的晶体管，成本恐怕也是超级高。建造64位计算机就属于这种情形，不但需要昂贵的晶体管，还同时需要两倍数量的晶体管。因此，工程师们一直探索在一般的技术架构下改善计算机的性能。

5.2.2 多处理

让计算机更加有效的一种方式是在同一时间执行多个程序。你可能正在写作业，然后停下来去加载一个网页，并且同时查看计算机正在自动接收的你室友发来的邮件，与此同时还有人正在下载你的个人主页看你近来的照片。只有一个CPU（因此只有一个指令寄存器），计算机如何应付这些工作？

除了再买一个电脑——这是可行的，但是这样很昂贵并且对技术要求很高——一种通常的选择是分时（time-sharing）。如同分时使用假日公寓一样，将时间分为时间片（假日公寓以星期为时间片，CPU以毫秒或微秒为时间片），允许使用设备的某个时间片。当这个时间片用完时，其他人轮换进来度假一周或使用CPU。为了以这种分时方式工作，计算机必须能将程序停止在任意处，将与程序有关的信息（栈的状态，局部变量，当前PC等）拷贝到主存中，并且从不同的存储区域加载其他程序的有关信息。只要时间片和存储区域是各自独立的（我们稍候将会看到如何实现这两点），计算机就好像在同时运行若干不同的程序。

出于安全的考虑，每个程序必须能够独立地运行，每个程序都应防止影响其他的程序。另一方面，计算机需要有一种系统化的方式能够适时地将用户程序换入换出CPU。这不是靠每个用户程序的良好表现，而是要借助操作系统进行管理。操作系统本身也是一个程序，但是操作系统的主要工作是控制其他的程序，并且实施安全规则。操作系统（简称OS，如MacOS、OS X，乃至MS-DOS）被授予普通用户级程序所没有的特权，包括中断现行程序的能力、向与程序使用无关的存储区域写入数据的能力等——这些权力常被形式化为编程模型，并且定义了超级用户与普通用户权限的差别。

5.2.3 指令集优化

加速计算机的一种方法就是使每条指令执行得更快些。例如，经常出现的指令应该在硬件上做些调整，使其比其他的指令执行得更快些。这种优化技术已经用在了JVM上，例如`iload_0`指令。`iload_0`指令比与其等价的`iload 0`指令长度短（1个字节）并且速度快。（当然，几乎每个方法都会使用局部变量#0，但是很少会用到比如说局部变量#245。）取决于要运行的程序，有些指令经常使用，设计者应该对其进行优化。

例如，在上面描述的多道程序系统中，“将所有的局部变量保存到主存”应该是一个常见的行为。一个更易理解的常见并且高需求应用的例子就是图形比重高的电脑游戏。良好的图形性能反过来需要快速将数据（位图）从主存移到图形显示设备。每次向CPU装载一个字的数据然后再将其存入显卡，不如用伪指令将一大块数据直接从存储器移至显卡。这种存储器直接访问（DMA）是很多现代计算机都支持的基本指令类型。类似地，对整个存储块执行算术操作（例如，用一个操作将全屏变为橙色）是Intel后期部分芯片基本指令集具备的能力。“对不同的数据独立执行同种操作”是对处理能力的根本性增强。借助于并行操作（这种并行操作称为SIMD并行，即单指令多数据）可以极大地提高程序的有效速度。

5.2.4 流水化

使CPU工作得更快的另外一种方式是在给定的微秒时间内打包处理多条指令。正如字面

表述的意义，一种可能就是在一个时间内处理多条指令。为了实现这一点，CPU需要有一个复杂、流水化的取指-执行周期允许其同时处理多条不同的指令。

等一下！这怎么可能？窍门在于多个操作依次进行，但是每个操作分为多个阶段，而且各个阶段以流水化的方式处理。举一个实例，一排人站成一个传水的队列，我不是将水从井口拎到40英尺远的炉灶旁（可能需要1分钟），而是将水桶从与我邻近的人手中接过来，然后传给离我4英尺远的下一个人。尽管水桶离炉灶还有36英尺远，但是我的手已经空闲下来了，可以接送下一桶水了。每桶水从井口传到炉灶仍需要1分钟的时间，但是可以有10桶水在被同时传递，因此单位时间里就有10倍数量的水被送到炉灶旁。另外一个好案例就是汽车生产线，每个工人不是一次组装全车，而是只负责单一的任务环节，汽车的组装由多个小环节构成。更普通的一个例子，如果我有很多的衣服要洗，我将一部分衣服放入洗衣机，当这些衣物洗完时，我又将其放入了干燥机，然后我又将另外一部分衣服放入洗衣机，这时两个机器会同时运行。

现代高端CPU都进行这种任务分解。例如，当CPU的一部分正在执行一条指令时，CPU的另外部分已经正在取其他的指令了。当这条指令执行完成时，下一条指令已准备就绪等待执行。这种流程也称为指令预取（instruction prefetch）；在CPU真正需要这条指令之前，它就被取出了，因此这条指令立即可用。本质上，CPU同时服务于两条指令，因此可以在给定的时间内处理两倍的指令，如图5-1和图5-2所示。这并不能改善延迟（latency）——每个操作从开始到结束仍然需要同样的时间——但是可以显著地提高吞吐率（throughput），即CPU每秒能够处理的指令总数。

	2p.m.	3p.m.	4p.m.	5p.m.	6p.m.	7p.m.
洗衣	负载1			负载2		
甩干		负载1			负载2	
折叠			负载1			负载2

图5-1 非流水化洗衣：6个小时内洗两次

	2p.m.	3p.m.	4p.m.	5p.m.	6p.m.	7p.m.
洗衣	负载1	负载2	负载3	负载4		
甩干		负载1	负载2	负载3	负载4	
折叠			负载1	负载2	负载3	负载4

图5-2 流水化洗衣：6个小时内洗四次

流水线的阶段数通常是不相同的，取决于具体的计算机，通常新的、速度快的计算机流水线的段数要多一些。例如，典型的中等级别的PowerPC（如603e型）采用一个4段流水线，可以同时处理最多4条指令。第一个阶段是*取指*阶段，将指令从主存载入CPU，作为下一条要被处理的指令。一条指令一旦被取出，*分派*阶段就对其进行分析确定该指令的种类，并且从相应的位置获取源操作数，然后准备由流水线的第三个阶段（*执行*阶段）执行。最后*完成/写回*阶段将计算结果送到相应的寄存器，并且根据需要更新整个机器的状态（图5-3）。

为了使流水化的过程尽可能有效，流水线应该一直都充满指令，并且数据一定要平稳地流动。首先，流水线只能以其最慢流水段的速度运行。简单的事情，如取某一条指令，能够和机器访存一样快地完成，但是指令的执行，特别是复杂的指令，就需要更多的时间。当其

中某条指令执行时，可能会在流水线中导致障碍（有时称为“气泡”），致使其他的指令堆积在其后，如同很多小轿车跟在一个慢吞吞的公交车后。理想的情况是每个流水段应该占用同样的时间，流水线的设计者应该确保做到这一点。

	1μs	2μs	3μs	4μs	5μs	6μs
取指	指令1	指令2	指令3			
分派		指令1	指令2	指令3		
执行			指令1	指令2	指令3	
写回				指令1	指令2	指令3

图5-3 类PowerPC的流水线

其他中断流水的方式就是从错误的存储器位置加载了错误的数据。这一考虑的最坏违反情况就是条件转移，例如 “jump if less than”（如果小于就跳转）。一旦遇到了这条指令，下一条指令要么来自于指令序列的顺序下一条，要么来自于转移目标处的指令——我们可能不知道是哪一条。事实上，我们没有办法知道是哪一条，因为条件依赖于流水线的计算结果，当前尚无法得到该结果。无条件转移的情况还不太坏，只要计算机有办法足够快地辨识它（通常在流水线的第一阶段进行）就行。子程序返回也带来其特有的问题，因为返回目标保存在寄存器中，可能已经不可用了。最坏的情况是，计算机别无选择，只能停下来等待流水线排空（这会严重影响性能，因为转移指令很常见）。考虑到这一缘由，很多研究开始关注预测转移目标的能力，以便继续保持流水线的填充。转移预测（branch prediction）是计算机猜测是否执行给定分支（以及转移目标）的艺术。基于这一猜测，计算机将继续执行指令，产生的结果也可能是无效的。这些结果通常保存在流水线的特殊位置，如果猜测被确认是正确的才将其拷贝到相应的寄存器中。如果猜测是错误的，这些位置将被清空，计算机在一个空的流水线上重新开始。可以想象，最坏的情况就是流水停顿，如果计算机猜测正确，还可以节省一些时间。

即使较差的算法也应该有50%的猜测是正确的，因为转移要么成功要么不成功。检查整个程序作出更准确的猜测通常是可能的。例如，对应for循环的机器代码通常包括一个代码块和一个位于循环结尾处的（向后）转移。既然大多数此类循环都执行不止一次（通常成百上千次），就要有很多次转移成功和一次转移不成功。这种情况下猜测转移成功可以达到99.9%的准确率。更加精确的分析是否看每个转移指令的历史。如果一个转移指令已经执行了两次，而且两次都没有转移成功，那就可以推测它这一次也不会转移成功。借助大量的多种可用信息，工程师们已经可以让现代处理器的流水化设计非常准确地（90%以上）猜测转移了。

5.2.5 超标量体系结构

涉及多条不同指令的其他技术还有重复设置流水阶段甚至是整条流水线。流水化的难点之一是保持各个流水阶段的均衡；如果某条指令的执行比其取指时间要长得多，那么就应该对流水阶段进行备份。超标量处理的思想就是在同一个周期立即执行多条不同的指令。为了充分理解这一点，需要将取指-执行周期一体化，并且假设并非每次只取一条指令，而是我们有一个等待处理的指令长队。（显然，这就是经常所说的指令队列——并没有假想的成份。）典型的CPU会采取模块复制来解决耗时的操作。一个类似的例子是给一个繁忙的高速公路增加一个车道，以便缓解增加的交通流量。类似地，想一下一般银行的处理方式，每个出纳员都可以为下一个客户服务。如果一个客户的问题比较复杂，需要花很多时间处理，其他出纳

员就可以缓解这种停滞。与先前描述的SIMD并行不同，这是一种MIMD（多指令，多数据）并行。当一条流水线正在处理某条指令（可能是浮点乘法）时，另外一条流水线可能正在进行另一个完全不同的操作（可能是加载一个寄存器）。

补充资料

Connection Machine

如果你想了解一个真正的并行操作，看一下Thinking Machines 公司在20世纪80年代末期制造的Connection Machine体系结构。CM-1（不久又有了更快的版本CM-2）型机包含了多达65536个不同的单元，每个都是一个独立的1bit处理器。所有的单元都与被称为“微控制器”的中央部件连接，这个微控制器向每个处理器发出相同的“纳指令”(nanoinstruction)。CM-5型机只能控制16384个不同的处理器，但是每个处理器都是一个功能强大的Sun工作站。这些处理器独立运行，但是也可以快速、灵活地将其互联，用来完成高速的并行计算。

最初的CM-1采用一种定制单元体系结构，每个芯片上有16个单元。这些芯片又被连接在一起形成一个12路的超立方体（12-way hypercube）用来构建一个非常密集的网络，并且所有的单元彼此可以快速地通信。理念上，Connecion Machines要尝试挖掘大规模并行的可能性，如人脑一样，并且超越传统冯·诺依曼体系结构的某些限制。一个神经元不具备强大的计算能力，但是正常人脑中10^{12}个神经元可以完成惊人的工作。以实用的术语描述，CM-1可以被看成是64K路SIMD并行。遗憾的是，专用芯片的成本太高，因此CM-5转向较少数量的商用SPARC芯片，并由此放弃了SIMD处理（像人脑一样）而青睐于MIMD。目前CM-5的后继机型在例如Beowulf机群并行处理中非常活跃。

5.3 存储器优化

为了使计算机尽可能快速、平稳地运行需要确保两件事情。首先，计算机所需的数据要尽快地可用，以使计算机不必耗时去等待。另外，存储器应该防止被意外地重写，比如用户邮件代理不能从Web浏览器上误读数据并将你所看到的该页内容发给其他的人。

5.3.1 cache存储器

对于32位字长的计算机，每个寄存器都能表示2^{32}（约40亿）个模式。理论上，这允许处理器使用4GB的存储器。而实际上，计算机所安装的存储器容量通常要小得多，对于某个程序来说实际使用的存储器容量就更小了。最重要的是，程序通常在某个时刻只使用整个程序中的很小一部分（比如，Web浏览器中下载网页的代码只有在你真正点击按钮时才会使用）。

存储器有多种不同的访问速度；也就是说，不同的存储芯片检索1比特信息所需要的时间是不同的。由于速度的关系，最快的存储器芯片的成本也最高。大部分程序在某一时刻只需要较小的存储容量，因此大多数实际的计算机采用多级存储结构。尽管CPU芯片本身可以达到2或3GHz的运行速度（即0.3~0.5ns内执行一条指令），但是大部分存储器芯片都非常的慢，有时存储器的响应时间是50~100ns。这看起来也许已经很快了，但是比起CPU芯片慢了将近400倍。为了减少存储器访问瓶颈，计算机也采用少量非常高速的存储器，但是容量要小得多（通常最多几兆字节），称为cache存储器。（读作“CASH”存储器，源自法语动词cacher，意思是“隐藏”。）它的基本思想是近期经常使用的存储区域被复制到cache存储器中以便CPU需要其中的数据时能够尽快地得到。cache存储器的设计和使用是一个艰巨的任务，CPU自身处

理这其中的细节，程序员不必为此担心。大多数计算机支持两级cache：一级（L1）cache位于CPU片内并且以CPU的速度运行；而二级（L2）cache是一组特殊的高速存储芯片，靠近CPU位于主板上。正如你所期望的，L1 cache速度较快并且昂贵，这意味着L1 cache容量最小，但是能够带来最大的性能提升。

5.3.2 存储管理

拥有32位字长的计算机能够向2^{32}个不同的存储单元写入。（当然，64位的计算机有2^{64}个不同的地址。）这些地址定义了程序可用的逻辑存储空间。当然，任何计算机可用的物理空间取决于所安装的芯片存储容量，也部分地取决于计算机的拥有者能够或愿意花多少钱购置存储容量。程序不指向某一物理存储位置，而是指向某一逻辑地址，然后由存储管理将其转换为某一物理存储器位置，甚至可能是硬盘的某一位置。

存储管理通常被认为是操作系统的功能，但是考虑到速度、便捷以及安全的因素，很多计算机对此提供了硬件上的支持。尽管考虑到安全问题使得用户级程序不能访问这部分硬件。这就意味着存储系统的大多数有趣的部分对于用户来说是看不到的，而只有管理模式下执行的程序才能感受到。用户所关心的存储器只是一个逻辑存储空间的一维数组，其中的每个元素都可以独立访问。因为这很重要，所以有必要再次强调：即使真实的物理存储器比逻辑地址空间相当大或者相当小，用户级程序都能够视逻辑地址如同物理地址，并且用适当长度的位模式表示物理存储器的某个存储位置。

在存储管理之下，需要进行逻辑存储地址到物理地址的转换。地址转换使得计算机能够访问直接与物理存储器对应的存储位置。这个过程利用了地址替代将一个地址空间（逻辑存储器）转换为另一个地址空间（物理存储器）。这种存储过程以及存储地址空间通常被称为虚拟存储。为了简明地解释这个问题，我们将以32位的PowerPC为例对有些抽象的存储管理加以介绍。有多种不同的方法完成这一转换，下面详细说明。

5.3.3 直接地址转换

最简单的确定物理地址的方法就是直接地址转换，当硬件地址转换已经被关闭时采用此方法（显然，只有管理程序能够将硬件地址转换关闭）。在这种情况下，物理地址的每一位都与逻辑地址相同，并且只能访问到4GB的存储空间。如果两个进程试图访问同一个逻辑地址，没有简易的方法来阻止。直接地址转换通常用于关心速度的专用计算机，每次只运行一个程序；除此之外，大多数操作系统实施了另外某种地址转换。

5.3.4 页式地址转换

为了防止两个进程访问同一个物理地址（如果允许地址转换时），CPU的存储管理系统实际上将逻辑地址空间扩展到第三个空间，称为虚拟地址空间。例如，我们可以定义一组24位的段寄存器来扩大地址值。在这种情况下，逻辑地址的高4位定义并选择一个具体的段寄存器。这个段寄存器里保存的数值定义了一个具体的24位虚拟段标识符VSID（Virtual Segment IDentifier）。这个虚拟地址由24位的VSID和逻辑地址的低28位并接而成，如图5-4所示。这就创建了一个新的52位地址，因此能够处理更多的存储进而防止冲突。

例如，计算机要访问存储位置0x13572468（一个32位地址）。高4位（0x1）指示计算机应该查看段寄存器#1。进一步假设这个段寄存器包含数值0xAAAAAA（24位）。将该值与最初存储器地址并接为一个52位的地址0xAAAAAA3572468。另一方面，如果另外一个程序要访问同一个逻辑地址，但是段寄存器内的值是0xBBBBBB，第二个程序的地址将是0xBBBBBB

3572468。因此，两个程序访问同一个逻辑地址，然而得到两个不同的VSID。这就解释了局部变量#1如何能在两个不同的程序中拥有不同的存储地址。

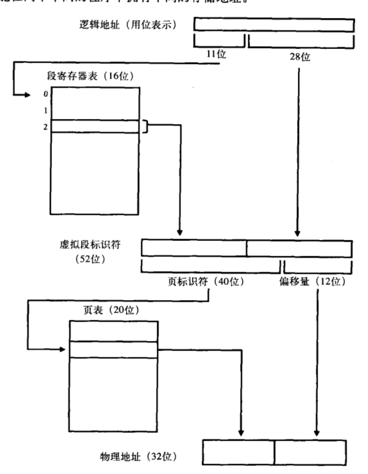


图5-4 理想的类PowerPC的虚拟存储结构图

当然，还没有计算机拥有2^{52}B存储器（即4PB）。目前的物理存储器实际上还要通过另外一个表来访问。物理存储器被分为页面，每个页面大小是4196字节（2^{12}B）。每个52位的虚拟地址可以被认为由40位的页标识符（这是一个24位VSID和一个从最初逻辑地址中截取的16位页标识符）和一个12位的页内偏移量构成。计算机保存一组页表，实质上是哈希表，用一个20位数作为每页的物理位置。通过查表，40位的页标识符被转换为20位的物理页地址。在这个过程的最后，最终的32位物理地址只是页地址再加上偏移量。

这并不像听起来那么令人困惑，尤其是当你看了图5-4以后。不过这里可能涉及了大量工作。计算机为什么要进行这样的过程？这样做有几个好处。首先，不是每个虚拟存储器的页面都需要被保存在物理存储器中。当页面不经常使用时，可以将它们换出，然后将它们保存在长久的存储设备中，例如硬盘。（这是谈论“虚拟存储”的最初原因，即计算机能够访问并非实际的内存单元，而是硬盘。这使得计算机以速度损失为代价来运行比物理存储空间大得多的程序。）另外一个好处就是通过改变段寄存器内的数值，同一逻辑地址可以指向不同的物

理地址。最后，基于按页处理的方式也在一定程度上增加了安全性。页表中的指定页能够标记为“管理独有”，这意味着只有管理级程序能够在该页面进行读出或写入。类似地一个页面也可以被标记为“只读”（程序能从该页面读出数据但不能向该页面写入数据）；“只有管理者写”（用户级程序能够从中读取数据，但是只有管理级程序才能写入）；或者“读/写”全都包含（任何程序可对该页进行读/写）。这将防止用户程序误写操作系统的重要数据。

5.4　外设优化

5.4.1　忙-等待问题

为了最佳地发挥外设的性能，外设一定不能阻止CPU做其他有用的事情。计算机如此的快速以至于它们几乎比任何其他的物理过程速度都更快。举一个简单的例子，一位熟练的打字员每分钟能键入120个字，这意味着大约每十分之一秒键入一个字符。一台1GHz的计算机在两次按键之间能够完成100，000，000个数的加法。因此，计算机应该能够在执行字处理程序的同时去做很多数字处理操作。但是计算机在进行操作时如何能够及时地响应键盘的请求呢？

处理这一问题的一个笨方法就是轮询，定期查看是否有事情发生。用高级伪代码可将这个操作描述如下：

```
while（没有键按下）
    等待一会儿
辨识出按下什么键并且做相关的事情
```

轮询不能有效地利用CPU，因为CPU不得不花费很多时间反复确定是否有事情发生了。这也是为什么轮询有时被称为忙-等待，因为计算机忙于等待按键而不能做任何其他有用的工作。

5.4.2　中断处理

处理期待中的未来事件的更聪明的方法是建立一个程序密切监视事件的发生并且做相关的处理。当事件发生时，CPU将中断当前的任务，然后利用先前建立的程序去处理该事件。

这就是大多数计算机如何处理期待但不可预知事件的一般做法。CPU建立多个不同种类的中断信号，中断信号由预先规定的事件产生，例如按键。当此类事件发生时，标准的取指-执行周期要有些变化。CPU不再加载并执行PC所指的下一条指令，而是查阅包含中断程序位置的中断向量表。然后将控制转移（就好像通过call或jsr转向一个子例程一样）到该位置，专用的中断处理程序将被执行以便完成所需的任何处理。（在具有标准编程模式的计算机中，这一控制也标志着从用户模式切换到管理模式。）在中断处理程序的最后，计算机返回（就像从子程序中返回）到主任务。

在大部分计算机中，可能的中断编号是从0到一个小的数值（比如10）。这些数与编入到中断向量表中的位置对应。当0号中断发生时，CPU就会跳转到中断向量表中的位置0x00，并且执行该处所存储的代码。如果1号中断发生，CPU则跳至0x01，依此类推。通常，向量表内的中断位置处保存的只是一条转移指令，用来将控制转移到真正完成实际工作的代码块上。

这种中断处理机制也可以推广到系统内部事件的处理。例如，CPU分时可以通过设立一个内部定时器进行控制（这类定时器如何工作的细节将在第九章讨论）。当定时器到达满计时，就会产生一个中断，致使计算机首先从用户模式切换到管理模式，然后跳转到中断处理程序，并由中断处理程序将当前程序环境上下文换出，然后换入接下来要执行程序的上下文。定时器然后可以重启，并且为新的程序恢复计时。

5.4.3 与外设的通信：利用总线

正如第一章所讨论，数据必定要在CPU、主存和外设之间通过一条或多条总线移动。这好比可通过多条路从你家到商店；行程可能快一些或者慢一些，它取决于道路的质量、司机的技能以及交通流量。无论是计算机还是购物者，都希望行程尽可能地快。

关于总线的典型使用有两个关键的问题。第一，从电气角度讲，总线通常就是将所有部件同时连在一起的一组线。这意味着总线担当了小规模的广播媒介，挂接在总线上的每个外设可以在同一时间获得同样的信息。第二，一个时刻只能有一个设备向总线发出信息；如果键盘和硬盘都要向总线发送数据，那么谁也不能成功发送。要成功地利用总线，各部分都需要遵循戒律。这个戒律以严谨的通信协议的形式表现。一个典型的总线协议可以包括CPU发出一个开始START消息，然后发出某个设备的标识。总线上的每个设备都能收到这两个消息，但是只有指定的设备会响应（一般会用ACKNOWLEDGE应答）这个消息。所有其他的设备都被这个START消息警示不能进行通信，直到CPU完成并且发出类似STOP停止消息。此时只有CPU和指定的设备允许使用总线，这样就减少了竞争和交通流量问题。

5.5 本章回顾

- 作为虚拟机，JVM不受基于物理芯片体系结构在设计和性能方面的实际影响限制。
- 随着芯片市场的竞争，工程师们已经开发了很多技术以便获取芯片的最佳性能。这包括在安全和速度两方面的改进。
- 一种获得用户级性能的方式是改善芯片的基本指标数字——大小、速度和延时，但这通常是一个困难并且昂贵的过程。
- 另外一种改善系统性能的方式（从用户角度）就是允许在一个时刻执行多道程序。通过分时，计算机能够做到这一点，这里程序在非常短的瞬间执行，并且程序会换入换出。
- 当工程师们知道在所设计的计算机上运行何种程序时，他们就能针对这些程序创建专用指令和硬件。计算机游戏就属于这样的程序，游戏程序对于计算机的图形处理能力有特别的需求。Pentium处理器就提供基本的机器级指令以便加速图形性能。
- 一个时刻执行多条指令的并行也能提高性能。在同时执行的指令种类方面，SIMD并行不同于MIMD并行。
- 取指-执行周期被分为若干阶段，每个阶段独立并且同时执行，这种被称为流水化的并行形式能够带来显著的性能增强。例如，在执行一条指令的同时去取下一条指令，即便存在延迟，计算机也能够获得对吞吐率100%的提高。
- 超标量体系结构将整个流水阶段复制若干次，借助于同时做若干同样的事情来提供另外一种加速处理的方式。
- 通过采用cache存储器加速访问经常使用的数据能够减少存储器访问时间。
- 存储器管理技术，如虚拟存储器和换页，能使计算机更快并且更安全地访问更大容量的存储器。
- 中断取得了显著的性能提高，避免了计算机检测是否有预期事件发生而浪费时间。
- 设计适合的总线协议能够加速计算机内部数据传送，减少对传输时间片的竞争。

5.6 习题

1. 实际计算机的哪些存储限制是JVM堆栈能够忽略的？

2. a. 128位的CPU与64位CPU相比有何优势？
   b. 这些优势有什么重要意义？
3. 一条保存主存栈全部内容并且从主存检索内容的专用指令对于JVM有帮助吗？
4. 如果由你负责，你打算对JVM体系结构做何增强？为什么？
5. 除了课本中提到的内容，给出两个实际的流水化示例。
6. 你如何将转移预测应用到lookupswitch指令？
7. 除了课本中提到的内容，给出两个超标量处理的实际示例。
8. cache如何决定保存什么内容以及删除什么内容？
9. 解释存储管理如何允许两个程序同时使用同一个存储位置却没有冲突？
10. 存储器映像I/O与虚拟存储系统是如何交互的？



第6章 Intel 8088

6.1 背景

1981年IBM发布了它的第一代个人计算机，此后得到广为应用并且各种机型被统称为IBM-PC。作为一个相对低成本的计算机，并且由家喻户晓的IBM公司生产（尽管Apple公司已经存在，但是只限于爱好者市场；当时几乎没有其他的计算机公司存在），它取得了巨大的成功，几乎立刻占领了微型计算机的购买市场。

当时计算机只有64K内存，没有硬盘，人们只能从5¼英寸软盘上向计算机加载程序。这个计算机内部的芯片是由Intel公司制造的，型号是8088。今天，8088的后代计算机仍然基于最初的8088设计。

从技术上讲，8088是第二代芯片，它基于先前的8086设计。两个芯片之间的区别很小；都是16位计算机，段式存储器体系结构。其主要不同在于8086有一个16位的数据总线，所以一个寄存器的全部内容可由一个总线操作送至存储器。8088只有8位数据总线，所以从CPU到存储器之间的存储或加载操作需要的时间要长一些，但是8088芯片的生产成本要便宜一些，这就降低了IBM-PC的整体价格（因此提高了其市场份额）。

随着基于8088的IBM-PC的成功，Intel和IBM有了一个稳固的市场并且改进了芯片。80286（尽管80186也有设计，但它在个人计算机市场上销售一直不看好）将安全特性融入了不安全的8088，并且运行的速度也比8088快很多。80286芯片是IBM-PC/AT（Advanced Technology，先进技术）的基础，也被称为PC/AT。

80386增加了寄存器的数量，并且寄存器的位数都变大了（每个都是32位），这使得芯片能力更强。随后的80486和Pentium（又名80586）又有进一步的改进，这将在第8章中详细讨论。这些后期的芯片，加上最初的芯片，通常被称为80x86系列。

从某种意义上说，8088只是一道历史风景；即便作为低成本的微控制器（例如，这类芯片可以算出如何烘烤面包，或者电梯应停在哪层），它仍可与其他基于更现代原理和技术的体系结构比美。然而，Intel 80x86系列的后代产品为了维护已有用户的利益都遵循向后兼容的原则。例如，在1995年，当Pentium发布时，上百万人正在他们已有的80486系统上运行软件。Intel设计者们没有强迫人们去购买新的软件以及新的硬件（这可能导致他们从其他的公司购买软件和硬件，如Apple），而是确保为486编写的程序仍然能够在Pentium上运行。由于这个决定在各个阶段都没有改变，这使得在1981年为IBM-PC编写的程序仍然能够在现代的P4上运行。当然，它们无法利用现代技术的改进之处，例如速度的增加（最初的PC以4MHz运行，而现代系统的运行速度是它的1000倍），改善的图形分辨率，甚至现代设备的使用，如鼠标、USB设备等。由于向后兼容的特性，理解8088对于理解现代Pentium体系结构仍然是重要的。

6.2 组织和体系结构

6.2.1 中央处理单元

在总体抽象层次上看，Intel 8088 CPU和大多数其他处理器非常相像，包括JVM（图6-1）。

数据保存在一组通用寄存器中，由取入到控制器并且在控制单元中被译码的指令对数据进行操作，并遵循ALU电路所定义的算术运算法则。不过存在一些微妙但很重要的差别。

第一个差别就是由于8088是物理芯片，它的能力（例如，寄存器的数量）是固定不变的。8088包含8个所谓的“通用寄存器”。不像JVM栈，这些通用寄存器并未以特别的方式被组织，它们以名字而不是编号命名。这些寄存器命名为

AX BX CX DX

SI DI BP SP

尽管它们被称为通用寄存器，但是大多数寄存器都与附加的硬件有关，以便某些操作执行得快一些。例如，CPU有专用的指令使用CX作为循环计数器，AX/DX是唯一用于优化整型乘法和除法的寄存器对。SI和DI寄存器支持高速存储器传输（SI和DI分别代表源索引和目标索引），而BP寄存器常被用于栈指令指向局部函数变量和局部参数。

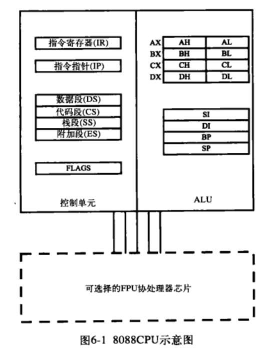


图6-1　8088CPU示意图

除了这些寄存器，8088还有一些特殊用途的寄存器，它们不用于一般的计算中。这些寄存器中最重要的是IP（指令指针）寄存器，IP保存将要执行的下一条指令的地址。（在其他的机器中，这个寄存器可能被称为程序计数器或者PC，它们的意义相同。）四个段寄存器（CS，SS，DS和ES）用于支持访问更多的存储器，并且使得对存储器的访问结构化。最后FLAGS寄存器保持一组单独的位描述当前运算的结果，如上一个操作结果是否为0或正数，亦或是一个负数。

这些寄存器都是16位宽。这意味着8088有一个16位的字宽，大部分操作能够进行16位处理。如果由于某些原因程序员想要使用小一些的表示，程序员可以使用通用寄存器的一部分。例如，AX寄存器可以被划分为两个8位的寄存器使用，它们被称为AH（高）和AL（低）寄存器（图6-2）。所有的子部分构成同一个完整的寄存器，所以任一部分的变化都会影响整个寄存器的数值。如果你向AX寄存器加载数值0x5678，这将会给AL赋值0x78，而给AH赋值0x56。类似地，此时将AL清零就会使AX寄存器的内容变为0x5600。这种划分对于所有的通用寄存器都成立，BX寄存器可以被划分为BH和BL寄存器，等等。

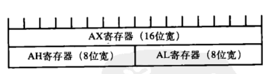


图6-2　8088中寄存器划分

8088寄存器中几乎所有的值都是以16位有符号补码形式表示的。不同于JVM，8088也支持无符号整型表示。在大多数操作中（例如，移动数据，比较两个位模式是否相同，甚至是两个数相加），8088并不关心给定的位模式是有符号还是无符号的量，但是少数情况下存在两种不同的操作。例如，两个无符号相乘使用助记符MUL；而两个有符号量相乘则使用助记符IMUL。

上述定义的寄存器用于8088计算机可完成的大部分操作，从存储器访问到控制结构和循环，以及高速整型处理。关于高速浮点处理，设计了一个独立的芯片作为浮点单元（FPU）。这个FPU有它自己的8个寄存器，每个寄存器80位宽用于高精度运算，并以栈的方式组织。保存在这些寄存器中的数据采用专用的浮点表示，与前面讨论的标准相似，只是尾数和指数的位数较多。还有一些附加的寄存器，两者都是FPU的一部分（例如，FPU有它自己的指令寄存器）并且用于某些专用运算。

6.2.2 取指-执行周期

8088的取指-执行周期与JVM的几乎相同；IP寄存器中的数值用于表示存储器地址，该地址（由IP寄存器指示）中的数值被拷贝（取）到指令寄存器中。指令一旦被取出，IP寄存器内的数值即增1（指向下一条指令），取出的指令由CPU执行。

唯一的限制就是IP寄存器自身的容量。由于只有16位宽，IP寄存器只能访问65000个不同的地址，因此最多取得65000条不同的指令。这看起来限制了程序的大小；没有程序能够超出64K。幸运的是，下一节描述的存储管理技术稍许缓解了这个限制，实际上是借助其他寄存器的附加位来扩大地址空间。

6.2.3 存储器

对于一个16位字宽的计算机，每个寄存器可容纳2^{16}（约65000）个模式。这意味着任何寄存器（例如，IP）拥有的存储器地址可以指向65000个不同的位置，以现代的标准看这绝对不够，简直枉费我们的努力。（来做一个快速的现实核查：排版这本书的文本处理软件占用了251864个存储器位置，这还不包括编辑和打印软件。）即使在1981年，64K字节存储器也被认为是非常小的容量。

关于这一问题有若干解决方案。最容易的方案是使每个模式/访问位置不指向单一的存储器字节，而是字（或者是更大的单元）。这在保存一个比字小的数据项（如字符串中的字符）时会降低效率，但是却使访问更多的存储器成为可能。

8088设计者采用稍加复杂的方法。8088的存储器被分割为64KB的段。每个段包含普通16位寄存器能最多访问到的存储器字节，但是这些地址被解释为相对基地址，基地址由特有的段定义。顾名思义，段定义保存在所谓的段寄存器中。

遗憾的是，数学在这一点上也很难处理。段寄存器本身是16位宽。真正的（有时称为绝对的）地址是由相关的段寄存器值乘以16（等价于左移4个二进制位置，或一个十六进制位置）再与相应的通用寄存器或IP中的值（称为偏移量）相加。例如，如果段寄存器中保存的值是0xC000，这将定义段从0xC0000开始。偏移量0x0000将对应（20位）位置0xC0000，而偏移量0x35A7将对应绝对位置0xC35A7，这些位置中的每一个都对应一个字节。

关于段的起始一定与64K的倍数对齐并没有特别的理由。加载数值0x259A将定义段的起始地址为0x259A0。事实上，任何一个以十六进制值0结尾的位置都是一个可能的段起始地址。在上述定义的段中，段值0x259A加上假定的偏移量0x8041会生成绝对地址（0x259A0+0x8041）或者字节位置0x2D9E1。为了简化起见，这样的地址对通常被写成段：偏移量的形式，例如259A：8041。在这种模式中，合法地址从0x00000（0000：0000）到0xFFFFF（F000：FFFF）。应该注意的是，通常有很多段：偏移量对指向给定的位置。位置F000：FFFF也是位置FFFF：000F，同样也是位置F888：777F（如图6-3）。

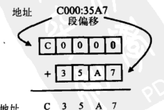


图6-3 段：偏移计算图示

按通常的习惯，四个段寄存器与不同的偏移量寄存器联合定义不同的存储类型及应用。它们对应于

- CS（代码段）：代码段寄存器与IP联合定义机器可执行的指令（程序代码）在存储器中的位置。
- DS（数据段）：数据段寄存器与通用寄存器AX，BX，CX和DX联合控制对全局程序数据的访问。
- SS（栈段）：栈段寄存器与栈寄存器（SP和BP）联合定义栈帧，函数参数和局部变量，下面将要讨论这个内容。
- ES（附加段）：附加段寄存器用于保持额外的段。例如，如果一个程序太大不能装在单独一个代码段中。

即使采用段式存储器模型，计算机也只能访问2^{20}个不同的位置，即1MB。即便这些存储空间在实际中也不是完全都可使用的，因为一些空间被预留给视频存储而不是通用的程序存储。实际上，程序员只能访问前512K用于程序的空间。幸运的是，当计算机存储器超出1MB在财力上成为可行的时候，8088的设计已经被同系列的后代产品超越了，包括Pentium。Pentium中存储器访问有了很大的不同，从某种意义上避免了1MB的限制。

6.2.4　设备和外设

现代计算机能够附带一批外围设备，从简单的键盘和显示器，到商业附件如扫描仪和传真机，以及真正的非普通专用设备如售货机、烤箱以及医学影像设备。由于基于80x86的机器如此普遍，因此设计并制成与8088接口的设备很常见。

然而，从计算机（以及计算机设计者）的视角，通过采用标准化的接口设计和端口，任务从一定程度上得到了简化。例如，很多计算机的主板上带有IDE控制器芯片，并且IDE控制器直接与系统总线连接。这个IDE控制器同时附带一组连线，这组连线可以插入IDE形式的驱动器。当该控制器从系统总线上获得相应的信号，它负责解释这些针对某个驱动器的信号。所有硬盘制造商都能完全确保它们遵从IDE规范并且适应任何PC。PCI（Peripheral Component Interconnection外围组件互连）规定了类似的标准化，将主板与各种设备直接连接。也许最广为使用的连接是USB（Universal Serial Bus通用串行总线）连接。这为任何支持USB的设备（从鼠标到打印机）提供了标准化的高速连接。USB控制器本身会查询关联其上的任何设备，发现它们的名称、类型以及它们要做什么（以及它们如何做）。USB端口也能提供能源，并且几乎以网络的速度与这些设备通信。然而，从程序员的视角，USB的优势是程序员只需编写程序与USB控制器通信，大大地简化了与设备通信的任务。

6.3　汇编语言

6.3.1　操作和寻址

8088在教科书中是一个CISC（Complex Instruction Set Computing复杂指令集计算）典型示例。CISC中能执行的操作集合庞大，并且操作本身的功能很强大。CISC设计的一个副作用是一些简单的功能能够使用不同的机器操作以多种不同的方式执行；这通常意味着Intel的设计者们为某些寄存器提供便捷使用的优化功能。

由于寄存器有名称，不像在JVM中使用编号或索引，基本运算格式就有所不同。在JVM中，只要简单地指明“add”就足够了；位于栈顶的两个数即被自动相加，并且在加法结束时，结果就被自动地留在栈顶。在8088中，情况就不同了；程序员必须指明加数被保存在哪里以

及和被存放在哪里。

由此，8088大多数指令采用两变量的指令格式，如下：

```
助记符        操作数1，操作数2；        可能的注释
```

通常第一个操作数被称为目标操作数（有时简写为dst），而第二个操作数被称为源操作数（简写为src）。基本思想是数据取自于源（源在其他情况下是不被改变的），而计算结果保留在目标中。因此，对于CX中的值与AX中的值相加（助记符：ADD）（并且将和保存在AX中），8088汇编语言指令将是

```
ADD    AX, CX              ; 真正的含义AX=AX+CX
```

由于有8个16位通用寄存器，所以有64（8×8）个不同的加法指令。事实上，既然可以将16位或者8位值相加，就会有很多的表示，例如

```
ADD    AH, BL              ; AH = AH + BL
```

8088还支持若干种寻址方式或者可以采用不同方式将给定的寄存器模式解释为访问不同存储位置。对于以上方面，最简单的例子就是寄存器中保存的值就是计算中所用的值。这被称为寄存器方式寻址。与其相比，也可采用立即方式寻址，这种方式将数据本身写入程序中，如

```
ADD    AX, 1               ; AX = AX + 1
```

注意，这只在1是第二个（源）操作数时才有意义。对目标操作数使用立即方式寻址将会导致错误产生。也应该注意到这是CISC计算的强势，因为这种操作在JVM中将占用两条单独的指令——一条指令加载整型常数1而另外一条指令执行加法。

立即数或者寄存器值也可以用做存储器地址，告诉计算机这不是数值而是要找到该值的地址。例如，如果BX包含一个8位数值的地址，我们用下面的表示将CL中的当前值与其相加。方括号（[]）代表寄存器BX使用间接方式（稍后将讨论直接方式）作为存储单元的地址：

```
ADD    CL, [BX]            ; CL=CL+用BX索引的存储单元中保存的值
```

这是一个难点，因此我们多做些研究。注意下面两条指令所完成的功能不同：

```
ADD    BX, 1               ; BX增1
ADD    [BX], 1             ; BX指向的值增1
```

它们如何不同？让我们假设存储在BX中的值是4000。执行第一个操作将使BX中的值增加，等于4001。相比之下，第二个操作没有改变BX的值（仍为4000），而是增加了保存在存储单元4000中的值。（实际上，这里有一个错误，原因将在6.4.1节讨论。本质上，尽管我们知道4000单元中有一个值，但是我们不知道它是字节，字，双字，还是其他的类型。因此计算机怎么能知道增加的单位大小呢？）类似地，执行

```
ADD    AX, [4000h]         ; 将4000h中的值与AX相加
```

查看存储单元0x4000（方括号中的数由于尾部的h被自动解释为16进制的量）中保存的16位的值，然后将其与AX中存储的值相加。在这个例子中，没有涉及寄存器而是采用了常数定义的存储位置，我们将此称为直接方式。

这些寻址方式能够用于几乎任意组合的ADD操作中，但是有两个例外。例如，将一个寄存器与另外一个寄存器相加，将一个存储单元与寄存器或者一个寄存器与存储单元相加，或者将一个立即值与寄存器或者存储器内的值相加都是合法的。然而，将一个存储单元与另一个存储单元直接相加是不合法的；其中的一个操作数一定要放在寄存器中。将立即值作为加法的目标地址也是不合法的。下面的一组操作中，前面的5个是合法的，而后面的2个是不合法的。

```
ADD   AX, BX          ; 寄存器-寄存器
ADD   AX, [4000]      ; 存储器-寄存器
ADD   AX, 4000        ; 立即值-寄存器
ADD   [4000], AX      ; 寄存器-存储器
ADD   [4000], 4000    ; 立即值-存储器
ADD   [3000],[4000]   ; 不合法！存储器-存储器
ADD   4000, AX        ; 不合法！立即值-任意
```

上述每个操作以及寻址方式都用略有不同的机器码表示。在机器码中，由于使用AX寄存器指示目标的8088特殊（而快速）指令的存在而增加了复杂性。聪明的程序员（或者编译器）利用这类指令将数据放在AX寄存器而不是其他寄存器能够使程序更快更短。（我们在JVM中已经看到了这类指令，即用iload_1作为更加通用的iload指令的捷径办法。难以理解的是，大多数汇编器对于高速的AX指令不使用有区分的助记符；而是汇编程序自身去识别这个特殊用途指令的使用，然后自动生成机器码。这意味着如果汇编器足够聪明汇编语言程序员就不必为这些指令而担忧。）这种为高速运算而设计的寄存器有时也被称为累加器，严格地说，8088有两个不同的累加器——或者至少同一个寄存器的两个部分。AX寄存器作为一个16位的累加器而AL寄存器用作一个8位的累加器。

6.3.2　算术指令集

对于大多数算数操作常见的是两操作数格式。例如，可用SUB指令完成两个数的减法而不是两个数相加：

```
SUB    BX, AX            ; BX = BX - AX
```

8088还支持存储器与寄存器之间的MOV指令，也同样使用两个参数的形式，其中第一个操作数是目标而第二个操作数是源。8088也使用同样的格式支持AND，OR和XOR指令；这些指令也都有专用的累加器捷径。

也有一些一个参数的指令，例如INC（自增），DEC（自减），NEG（求反——即乘以−1）和NOT（参数所有的位变反，等价于与一个各位全为1的参量做XOR运算操作）。格式非常简单：

```
DEC    AX                ; AX = AX - 1
```

乘法和除法都比较复杂。原因很简单：当你要两个16位整数相加（比如说）时，大约会得到一个16位的结果，该结果仍可以装入一个16位的寄存器。当你要将这两个数相乘时，结果就可能长达32位（一个16位的寄存器无法容纳）。类似地，整数除法实际产生两个结果，商和余数（例如：22/5=4余数2，就像小学算术）。因此，8088为了乘法和除法的结果增补附加的寄存器——由于硬件的复杂性，8088只能使用一些针对这些操作特定的寄存器。

要将两个数相乘，第一个数一定要出现在累加器（适于任何位宽）。乘法指令本身（MUL）有一个参量，该参量是乘法的第二个数。乘积被放在一个寄存器对中，确保乘法不会溢出。表6-1示意乘法操作中信息向哪里流动以及如何流动。

表6-1　8088乘法信息流

乘数	被乘数	乘积的高半部分	乘积的低半部分
AL	参量	AH	AL
AX	参量	DX	AX

（DX-AX寄存器对有时被缩写为DX:AX；AH:AL对通常缩写为AX寄存器。）

将两个数59和71作为16位值相乘，可用的代码片断如下。首先，（十进制）值59通过MOV指令被加载到AX寄存器，值71被类似地加载到BX寄存器。然后累加器（AX）中的值与BX中的值相乘。结果将被留在DX:AX。特别地，AX将保存结果的低16位（十进制值4189，保存为0x105D），而DX保存高16位，在该例中将是全0。

```
MOV    AX, 59                  ; AX获得乘数
MOV    BX, 71                  ; BX获得被乘数
MUL    BX                      ; DX:AX=AX*BX
```

实际上有两种不同的乘法指令，一种用于无符号整数乘法（MUL）而另一种用于有符号整数（IMUL）。两种情况中寄存器用途是相同的；唯一区别是被乘数和乘数的最高位被当作符号位还是数值位。例如，值0xFF（作为8位的量）要么是-1（作为有符号量）要么是255（作为无符号量）。用0xFF与其自身相乘会导致0xFF01存放在AX寄存器，而IMUL指令将得到0x0001（因为-1与其自身相乘结果当然是1）。

除法使用同样复杂的寄存器集，但是次序相反。被除数被放入寄存器对中，要么是AH:AL对，要么是DX:AX对。除法的参量用作除数。商被保存在寄存器对低半部分而余数被保存在高半部分。使用先前的例子：

```
MOV DX, 0                      ; DX清零
MOV AX, 22                     ; DX:AX=22
MOV BX, 5                      ; 除数是5
DIV BX                         ; 执行除法
                               ; 现在DX包含2
                               ; 现在AX包含4
```

类似于乘法，除法也有两种指令，DIV和IDIV，分别执行无符号和有符号除法。

6.3.3 浮点运算

8088 FPU几乎是单独的自包含计算机，有它自己的寄存器和特殊的指令集，专门用来执行浮点操作。实际上它是一个单独的自包含的芯片，以型号8087作为数学协处理器出售。数据在8088主CPU和协处理器之间传送。未命名的8087寄存器是8个80位宽的存储单元组成的栈（是的，就像JVM），再一次地，类似于JVM，指令集指明操作类型以及数据表示类型。

FPU能够保存和执行3种不同类型的数据：标准的IEEE浮点数，整数和专有格式称为二进制编码的十进制数（BCD），BCD中每4位一组代表一个十进制数字的二进制编码。这个格式常被IBM主机采用，因为对于工程师们很容易将这个二进制模式重新解释为（直观的）十进制数字，反之亦然。在8087内部，所有这些格式都被转换成80位的格式并且以80位格式保存，这在本质上比IEEE定义的标准格式更为精确。

所有的FPU操作都以字母F开头（JVM程序员应该熟知这一点）并且如人们所期望的对栈顶进行操作，就像JVM或者逆波兰计算器。FADD指令弹出FPU栈顶两个元素，将它们相加，然后将结果压入栈。其他的算术操作包括FSUB，FMUL和FDIV，其功能如所期待的那样。有两个附加操作：FSUBR和FDIVR，其执行反向的减法（除法），将第二个元素减去栈顶元素而不是从栈顶减去第二个元素。

补充资料

其他FPU变量格式

尽管FPU总是使用内部80位“扩展精度”格式进行操作，数据在主存中却以多种形式取/存。在取/存时进行转换，取决于所用的操作助记符。有很多不同的指令，大多数都有几种解释，如下：

- 整数：用FILD指令取16位或者32位整型数据。在后来的机器中，这条指令也能取64位的量。
- BCD整数：用FBLD指令取80位的二进制编码的十进制格式整型数据。
- 浮点数：用FLD指令取32位，64位或者80位长的浮点数。

这些量一旦取出，它们就被FPU视为相同。要存一个值，在上述的助记符中用“ST”代替“LD”。

还有一些特殊指令（如FSQRT）处理常用的数学函数，如平方根和三角函数。

用FLD指令能将数据压入（取入）FPU栈。这实际上以3种形式表现：FLD从指定的存储单元取32位或64位IEEE浮点数，FILD从存储器取16位或32位整数（并将其转换为内部浮点数），以及FBLD从存储器取80位BCD数（并对其进行转换）。有一些使用常数的特殊操作：FLD1取1，FLDZ取0，FLDPI取80位表示的π，还有一些其他的指令指定常用对数，例如2的自然对数。要将数据从FPU移入存储器，采用FST指令的一些变形——再次说明，有对整型（FIST）和BCD（FBST）存储的变形。一些操作还有额外的变形，尾部以P标记，当操作完成时弹出栈（例如，FISTP将栈中的浮点数弹出并保存为整数）。FPU的一个限制是数据只能从存储器取或者存至存储器，不能直接从ALU寄存器取。因此以下语句

```
FILD   AX                       ; 不合法！不能使用寄存器
```

就是不合法的；存放在AX中的值应首先移入一个存储器字位置然后再从该位置取出如下：

```
MOV    Location,AX              ; 将AX内容移入存储器
FILD   Location                 ; 从存储器取入FPU
```

6.3.4 判定和控制结构

如同大多数汇编语言，8088的控制结构建立在无条件和条件跳转指令基础上，在这里控制被转移到源码中声明的某条标号语句。如同JVM，这一处理实际上通过偏移量计算以及将程序计数器当前的地址与偏移量相加/减完成。跳转指令（助记符：JMP）的格式对于我们来说也是熟知的：

```
LABEL: JMP    LABEL                  ; 愚蠢的无限循环
```

条件跳转使用CPU中FLAGS寄存器中的一组二进制标志位（flag）。这些标志都是用独立的位来描述最近计算结果；例如，当且仅当最近操作（ALU中）结果是零时，零标志ZF被置位。符号标志SF包含最高位（如果结果是有符号整数该位将是符号位）的副本并且当且仅当最近结果是负数时该位被置位。当且仅当最近计算产生一个进位时，进位标志CF被置位，（当执行无符号数计算时）CF标志计算结果太大寄存器无法容纳。溢出标志位OF处理类似的情况，这里是指当执行有符号数计算时对于寄存器来说太大（或太小）。还有一些其他的标志位，但是上述例举的都是最常用的标志位。

条件跳转的助记符形如“Jcondition”，这里“condition”描述跳转成功时的标志设定。例如，JZ的含义是如果零标志被置位则跳转，而JNZ的含义是如果零标志未被置位则跳转。我们可以以此来测试两个值是否相等：

```
        SUB    AX, BX                  ; AX = AX - BX
        JZ     ISEQUAL                 ; 如果现在AX=0则设置ZF (=1)
        ; 如果到了这里，AX就不等于BX
        JMP    OUTSIDE                 ;
ISEQUAL:
              ; 如果到了这里，AX等于BX
OUTSIDE:
              ; if/else语句之后返回
```

其他的条件跳转包括JC/JNC（如果CF置位/清零则跳转），JS/JNS（如果SF置位/清零则跳转），JO/JNO（如果OF置位/清零则跳转），等等。遗憾的是，不是所有的标志都有这么清晰的算术解释，另一组条件跳转是处理算术比较，如“大于”，“小于或者等于”，等等。这些指令采取与算术关系相适的组合方式来解释标志。

更加详细地说，这些跳转期望标志寄存器包含一个数减去另一个数的结果，就像上面刚

举例给出的示例片段（这是一个小谎言，稍后会更加详细解释CMP指令）。要确定一个有符号数是否比另一个大，可以使用JG（如果大于则跳转）助记符。其他助记符包括JL（如果小于则跳转），JLE（如果小于或者等于则跳转）以及JGE（如果大于或者等于则跳转）。这些助记符也以取负的形式存在——JNGE（如果不大于或者等于则跳转），当然，这等价于JL——事实上用两个不同的助记符实现了同一指令。类似地，存在JE，等价于先前定义的JZ，JNE与JNZ相同。

对于无符号整数的比较，需要一组不同的指令。要理解其中的原因，考虑16位的量0xFFFF。作为有符号整数，它代表-1，小于0。作为无符号整数，它代表最大的16位数，略大于65000——当然它大于0。所以问题“0xFFFF>0x0000？”有两个不同的答案。取决于该数是否为有符号数。8088考虑到这个问题，提供了一组条件转移指令（即JA，JB，JAE，JBE，JNA，JNB，JNAE，JNBE）用于比较无符号数。所以，要判定AX中保存的值能否装入8位寄存器，使用下面的代码片段：

```
        ; 8位安全测试版本1
        SUB     AX, 100h                    ; 从AX中减去2^8
        JAE     TOOBIG                      ; 无符号比较
        ; 如果程序到这里，说明数据能装入8位寄存器
        JMP     OUTSIDE
TOOBIG:
        ; 如果程序到这里，说明AX中的数据超过8位
OUTSIDE:
        ; 继续做所需要的工作
```

这个代码段的一个问题是，为了正确设置标志，AX的值必须被修改。一个可能的解决方法是要保存AX的值（MOV该值到存储单元）并且在设置标志之后再取回该值。这是可行的，MOV指令不影响标志寄存器，能保持先前的设置。因此，我们可以重写8位安全测试，该测试稍微多用些空间和时间，代码如下：

```
        ; 8位安全测试版本2
        MOV     SOMEWHERE, AX               ; 将AX存入SOMEWHERE
        SUB     AX, 100h                    ; 从AX中减去2^8
        MOV     AX, SOMEWHERE               ; 恢复AX存入，不影响标志位
        JAE     TOOBIG                      ; 无符号比较
        ; 如果程序到这里，说明数据能装入8位寄存器
        JMP     OUTSIDE
TOOBIG:
        ; 如果程序到这里，说明AX中的数据太大
OUTSIDE:
        ; 继续做所需要的工作
```

Intel指令集用一个专用命令提供了一种更好的方法。具体地，助记符CMP执行非破坏性减法。这条指令计算第一个数减去第二个数的结果（如上述例中所做），并据此设置标志寄存器，但是不在任何位置保存减法的结果。因此，重新将第一个版本改写如下：

```
        ; 8位安全测试版本3
        CMP     AX, 100h                    ; 比较2^8与AX
        JAE     TOOBIG                      ; 无符号比较
        ; 如果程序到这里，说明数据能装入8位寄存器
        JMP     OUTSIDE
TOOBIG:
        ; 如果程序到这里，说明AX中的数据太大
OUTSIDE:
        ; 继续做所需要的工作
```

该代码保持了第一版本的效率而又未破坏寄存器中的值。

除了这些相当传统的比较和转移指令之外，Intel提供了一些专用指令（有些重复，不是吗？）支持有效的循环。CX寄存器协调这一目的。具体地，借助JCXZ指令，如果CX的值是0计算机就跳转。借助该指令，可以建立计数器控制的循环

```
        MOV     CX, 100                          ; 循环100次
BEGIN:
        ; 做些有趣的事情
        DEC     CX                               ; 计数器减1
        JCXZ    LOOPEXIT                         ; 如果完成（CX=0）则退出
        JMP     BEGIN                            ; 返回开始处
LOOPEXIT:
        ; 现在在循环之外
```

甚至更简洁地，LOOP指令处理递减和转移——它将递减循环计数器并且如果计数器未达到0时转向目标标号。因此，我们能将上面的循环简化为一条有趣的语句：

```
        MOV     CX, 100                          ; 循环100次
BEGIN:
        ; 做些有趣的事情，当CX=0时退出
        LOOP  BEGIN                              ; 返回开始处
        ; 现在在循环之外
```

使用浮点比较结果就难以处理了。基本问题在于标志寄存器位于主CPU（具体在控制单元）内，CPU通过控制单元中的PC处理转移指令。与此同时，所有的浮点数都存放在FPU，FPU是与CPU完全独立的芯片。数据必须从FPU移入标志寄存器，这可通过一组特殊指令完成，这已经超出了讨论的范围。

补充资料

好的。如果你坚持还可以继续讨论。FPU提供FCOM指令和FTST指令，FCOM比较栈顶的两个元素，而FTST将栈顶元素与0相比。这个比较结果保存在“状态字”中，等价于标志寄存器。要使用这个信息，数据要移动，首先移入存储器（因为FPU不能直接访问CPU寄存器），然后再移入寄存器（因为标志寄存器不能直接访问存储器），最后送到最终目标。完成第一步的指令是FSTSW（保存状态字，将存储位置作为它的唯一参量）；第二步用普通的MOV将状态字移入AX足已；第三步使用专门的SAHF（将AH存到标志寄存器）指令。一般的无符号条件跳转就可以正确工作了。这一过程的复杂性解释并阐明了编写计算机程序时使用整型变量速度快的部分原因。

6.3.5　高级操作

8088提供了很多操作，我们只能涉及其中的一部分。它们中的许多，也许是大多数，是执行常见任务的捷径，比起使用简单指令只需用较少的机器指令。XCHG（交换）指令就是一个实例，它将源和目标参量的内容交换。另一个示例是XLAT指令，它使用BX寄存器作为表索引并且利用表中保存的量调整AL中的值。本质上，XLAT是操作AL=[[AL]+BX]的简写，它通过几个步骤执行先前描述的一些基本操作。（如果你需要将一个字符串快速转换为大写表示，这将会很有用）。

8088也支持对于字符串和数组类型的操作，为此要使用（另外一组）专用指令。我们将在6.4.4节看到它们的一些使用，因为字符串和数组通常被存放在存储器中（寄存器容量不够大）。

就像我们将要看到的，使用更多操作的趋势在系列中的后继成员中呼声更高，于是指令集变得愈发庞大并且复杂，到了一种目前几乎没有程序员知道Pentium4全部指令细节的程度。

（这正是编写一个好的编译器既重要又有难度的原因。编译器愈聪明程序员就可以愈愚笨。）

6.4 存储器组织和使用

6.4.1 地址和变量

8088存储器的段组织已经描述过。每个可能的寄存器模式代表存储器中一个可能的字节位置。对于较大的模式（一个16位字，或者32位“double”，甚至是80位“tbyte”，包含10个字节并且保存一个浮点值），人们能够增补两个或者多个相邻字节单元。只要计算机能够计算出有多少个字节在使用，访问不同容量的存储单元就非常简单了。

遗憾的是，这个信息对于计算机而言并不总是可用的。先前的小谎言暗示：

```
ADD     [BX], 1                 ; 增加BX所指向的存储单元内的值
```

是合法的。可惜存储在BX中的值是一个地址；因此没有容易的方法知道目标量是16位还是8位。（类似地，我们不知道需要加0x0001还是0x01。）取决于目标的容量，这个语句可以被解释/汇编为任何3种不同机器指令之一。汇编器（和我们）需要一个提示以便知道如何解释[BX]。这也同样适用于直接使用一个具体的存储地址的情况，如下：

```
ADD     [4000h], 1              ; 增加0x4000存储单元内的值
```

有两种方法给计算机一个这样的提示。简单点但是不很有用的方法是在语句中确切解释其含义，4000h应该被解释为指向一个16位的字地址，该行重写成：

```
ADD     WORD PTR [4000h], 1   ; 增加0x4000存储单元内的值
```

与其相对比，使用BYTE PTR将4000h强迫解释成1个8位地址，使用DWORD PTR（双字）将4000h强迫解释成1个32位地址。

一个更加通用的解决方法是提前通知汇编器，告诉汇编器要用一个特殊的存储单元以及希望使用的容量。这样一来，这个存储单元的名字可以像直接方式寻址那样被用作该存储位置内容的简略表示。这与高级语言如Java，C++或者Pascal声明变量的作用相似。取决于8088汇编语言的版本和生产者，下面的每一条都可行，定义了16位变量（名字由程序员选择）：

```
example1      WORD    1000    ; 1000
example2      DW      2000h   ; 2000h == 0x2000
```

现在这些值能以直接寻址方式随意使用：

```
MOV     AX, example1            ; AX现在是1000
ADD     example2, 16            ; example2现在是2010h
CMP     AX, example2            ; AX大于example2吗？（否）
```

这个声明用于几个目的。首先，计算机现在知道两个16位存储单元已经被保留用于程序中的数据。其次，程序员已经解除了准确记住这些存储单元的负担，因为程序员可通过有含义的名字引用它们。最后，汇编器已经知道它们的容量并且在编写机器代码时知道如何使用这些存储单元。这并不是说程序员不能超越汇编器

```
exple3 WORD     1234h           ; 定义16位空间
ADD     BYTE PTR exple3, 1      ; 合法但是愚蠢
```

但是这很有可能导致程序出现故障。（相比之下，如果你试图访问所存储的数据元素的某个部分，JVM就变得非常恼火并且通常不允许你这样做。）

汇编器将会接受多种类型的存储器预留，包括一些很大的类型以至于不能在寄存器中轻松处理。使用更加现代的语法，下列任一声明都是合法的定义：

```
Tiny    BYTE    12h             ; 单一字节
Small   WORD    1234h           ; 两个字节
```

```
Medium DWORD  12345678h        ; 四个字节
Big    QWORD  1234567812345678h
                               ; 八个字节
Huge   TBYTE  12345678901234567890h
                               ; 十个字节
                               ; (用于FPU)
```

浮点常数可以用REAL4（对于32位数）、REAL8（对于64位数）或REAL10（对于80位数）定义。数值本身通常以正常或者指数形式表示：

```
Sqrt2  REAL4  1.4143135      ; 实数
Avogad REAL8  6.023E+23      ; 指数表示
```

通过用？作为初值，可以定义存储单元而不必初始化。当然，在这种情况下存储器仍将包含某种模式，但是无法预料其中的内容，如果你试图使用那个值可能会发生错误。

6.4.2 字节交换

存储器中的内容如何与寄存器中的内容进行比较？具体地，如果16位模式0100 1001 1000 0101保存在AX寄存器中，它与保存在存储器中同样的16位模式的值相同吗？

出乎意料地，答案是“不相同”（也许它并不是那么让人意外，如果真的那么简单，这本书的这一节就不会存在了）。再者，作为一个古老机器的产物，8088具有相当奇特的存储模式。

人们写数时通常使用所谓的大端（big-endian）表示。所谓的最高有效数（对应最高次方的数）先写（和先存），而小一些的较低有效数跟在后面。快速检查说明：通常在纸上写数时按大端格式；首先写出的数字意味着基数的最高次方。类似地8088 CPU在寄存器中按大端排序存储数值。如上写出的值（0x4985）将代表十进制值18821。然而，存储器中的数据实际上按小端格式逐个字节保存。具有较小地址（0x49,0100 1001）的字节是最低有效字节；另一个字节是最高有效字节。（威廉姆小姐，我七年级的英语老师，坚持认为只有两个字节时不能说“最高”有效字节，只能用“较高”有效字节。但这种场合下专业术语会战胜传统英语语法。）因此，在存储器中这一模式将表示0x8448，几乎两倍大。这一模式也适应更大的数，于是32位的量0x12345678将存储为4个独立的存储器字节，如图6-4所示。

0x78	0x56	0x34	0x12

图6-4 0x12345678的存储，字节交换图示

幸运的是，程序员几乎不必记住这一点。任何时候数据移入或者移出存储器，字节交换都在硬件一级自动完成。除非程序员明确地无视汇编器关于数据大小的认识（如上节），唯独变得重要的时候就是处理多组数据，例如数组及其扩展、串或者从一种机器到另外一种机器移动数据。（当然，几乎不并不等于决不，当它变得重要时，这可能就是致使你要用头撞墙的某类错误的根源。）

6.4.3 数组和串

用于预留单个存储单元的表示也能用来预留大的、多单元的存储块。可以通过用逗号分隔的值列表或者简写DUP代表的重复值来设置存储值。例如，

```
Greet  BYTE   48h,45h,4Ch,4Ch,4Fh  ; ASCII "HELLO"
Thing  DWORD  5 DUP  0x12345678    ; 5个相同的值
Empty  WORD   10 DUP (?)           ; 10个空位置
```

分别定义了5个字节的字母（H，E，L，L和O）数组Greet，一组双字Thing，以及具有10个2字节数值的数组Empty，每个值均未初始化。

要访问这些数组元素，程序员需要从数组基地址开始索引或者偏移。如果Greet中的H被

存储在3000h，那么E将被存储在3001h，3002h处是第一个L，O存储在3004h。当然，我们不知道Greet究竟被存储在哪个地址，但是无论它被放在哪里，"H"要比"E"低一个单元的位置。通常，数组名本身指向数组开始值的位置。因此，要将前三个字母装入AH，BH和CH（字节）寄存器，可以使用

```
MOV    AH, [Greet]                ; 装入H
MOV    BH, [Greet + 1]            ; 装入E
MOV    CH, [Greet + 2]            ; 装入L
```

（至少，对于本书）这实际上是一个新的寻址方式，称为索引方式。如前，[X]表示"存储器X单元的内容"，但是在此例中，X是一个相当复杂的值，计算机需要实时地计算此值。由直觉可得[Greet]与[Greet+0]以及Greet自身相同，[Greet+1]是邻接的字节。由于计算机知道Greet包含字节（由存储器预留声明所定义），它假定[Greet+1]也是一个字节。

我们怎样访问Empty数组元素？第一个元素就是[Empty]，或者[Empty+0]，或者就是Empty本身。不过在这种情况下，由于Empty包含的是WORD对象，所以下一个字就不是[Empty+1]而是[Empty+2]！与大多数高级语言不同，索引的计算自动地考虑了元素的类型，8088索引方式寻址需要程序员处理元素空间大小的问题。

索引方式寻址涉及寄存器更广泛的用途。在高级语言中，例如，最常见的数组操作就是使用变量索引——如，访问循环内的元素a[i]，i是一个整型变量。等价的汇编语言利用一个通用寄存器作为索引表达式的一部分。表达式[Greet+BX]将访问数组Greet的第BX个元素（如果这个字有意义）。通过调整BX中的值（假设从0到4），表达式[Greet+BX]将依次选择每一个元素。类似地，按每次加2调整BX中的值，即按字的空间大小，我们可以用下述程序段将整个Empty数组初始化为0：

```
        MOV     CX, 10              ; Empty中有10个元素
        MOV     BX, 0               ; 从[Empty+0]开始
BEGIN:
        MOV     [Empty+BX], 0       ; 将这个元素置0
        ADD     BX, 2               ; 移向下一个字
        LOOP    BEGIN               ; 循环，直到CX=0退出循环
```

只有几个16位寄存器能够合法地用作这类表达式中的索引，8位寄存器都是不合法的。只有BX、BP、SI和DI寄存器能够使用；而且，BP寄存器还不能用于该用途，因为这样可能会出错。BP寄存器已经被操作系统留用，稍后将在6.4.7节讨论。

有经验的C和C++程序员也许已经急于找寻一种快速并且有效的方式了。不用每次经过循环时去计算[Empty+BX]，为什么不将BX本身设置为[Empty+0]所在的位置呢？依此建议得到的代码如下：

```
        ; 警告，这行不通！
        MOV     CX, 10              ; Empty中有10个元素
        MOV     BX, Empty           ; 从[Empty+0]开始（错误！）
BEGIN:
        MOV     [BX], 0             ; 将这个元素置0
        ADD     BX, 2               ; 移向下一个字
        LOOP    BEGIN               ; 循环，直到CX=0退出循环
```

尽管这是个好主意，但执行起来就不是那么回事了，主要因为MOV BX，Empty并没有按我们希望的去做。汇编器将Empty视为直接方式寻址的简单字节变量，并且试图将Empty的首字节（"H"）移入BX。这不是我们想要的。事实上，它甚至还不合法，因为我们试图将一个单字节移入一个四字节的寄存器中，这会导致空间大小的冲突。要明确地使用一个指针型变

量，我们借助关键字OFFSET产生存储器地址而不是其内容。

```
        ; 改进的可行版
        MOV    CX, 10              ; Empty中有10个元素
        MOV    BX, OFFSET Empty    ; 从[Empty+0]开始
BEGIN:
        MOV    [BX], 0             ; 将这个元素置0
        ADD    BX, 2               ; 移向下一个字
        LOOP   BEGIN               ; 循环，直到CX=0退出循环
```

当然，所得的实际时间/空间改进可能是有限的，因为用于索引的加法仍然由一条单独的8088机器指令完成。然而，特别是在紧凑的且经常执行的小循环中，每个小的改进都是有益的。

6.4.4 串原语

如同前面的Greet示例，串可以用字符数组实现（通常作为字节，有时也会更大些）。8088也提供一些串原语操作指令，这些指令易于快速地执行常用的串功能。这些基本操作都要使用SI，DI或者两者同时使用——SI和DI专门用于优化这些操作。

现在，我们就集中关注从一个位置到另一个位置的复制或移动这样的简单任务。依据串元素的基本大小，有两个基本的操作：MOVSB（移动字节串）和MOVSW（移动字串）。这种基于串元素基本大小的划分适于所有串原语操作指令；这里还有两点不同就是分别以B和W结尾。因此，为了简化解释，我们将相似的操作行为归结为MOVS？名下。

MOVS？操作将数据从[SI]复制到[DI]。这个操作本身只复制单独一个元素，但是很容易将其放到一个简单的循环体内。串原语操作的优点是CPU在机器指令中支持自动循环，在汇编语言级用一个前缀助记符来表达。最简单的例子就是用REP前缀，例如：

```
REP    MOVSB
```

这一行为很像LOOP？指令，在此CX寄存器被当作计数器。每次指令执行时，SI和DI的值都要调整，CX的值递减，指令反复地执行直到CX降为0。

有两种调整SI和DI的方式。首先，自动更新的幅度与MOVS？指令中所指元素的大小相对应，SI/DI的改变幅度对于字节指令是1，而对于字指令是2。其次，标志寄存器中的一个特殊标志（方向标志）控制地址从低到高（增大SI/DI）或者从高到低（减小SI/DI）。这个标志有两条指令控制，如表6-2所示。

表6-2 基本串的方向标志操作

指令	标志状态	SI/DI操作	地址序列
CLD	清零（=0）	加	低到高
STD	置1（=1）	减	高到低

要明白这是如何工作的，我们来设置两个数组，并在它们之间进行复制。

```
Arr1   WORD   100 DUP (-1)    ; 源数组：100个字
Arr2   WORD   100 DUP (?)     ; 目的数组：也是100个字

MOV    SI, OFFSET Arr1        ; 设置源地址
MOV    DI, OFFSET Arr2        ; 设置目的地址
CLD                           ; 清零方向标志
                              ; 从Arr1[0] 到Arr1[99]
MOV    CX, 100                ; 循环处理99个字
REP    MOVSW                  ; 进行拷贝
```

另外一个常见的操作就是比较两个串是否相等。这可用CMPS？操作完成，该操作隐含执行从源操作数减去目的操作数。重要的安全提示：该操作是CMP指令的相反过程，CMP是从

PDG

目的减去源！更重要地，该指令通过设置标志位使得正常的无符号条件跳转将完成应该做的事情。

REP前缀的另一个变形在这样的背景下特别有用。只要CX不为0并且零标志被置位REPZ（或者REPE）就循环执行。这实际上意味着“还没有到达串的结尾并且到目前为止两个串是相同的”，因为零标志仅当减法的结果是0时才置位，这意味着这两个字符是相同的。利用下面的代码段我们能够进行串的比较：

```
MOV     SI, OFFSET Astring    ; 设置源地址
MOV     DI, OFFSET Bstring    ; 设置目的地址
CLD                           ; 清零方向标志
MOV     CX, 100               ; 循环100次
REPE    CMPSW                 ; 比较
```

这段代码执行后会有两种可能发生。一种可能是：CX变为0且Z标志置位，这种情况下两个字串相同。因此，语句

```
JE      STRINGSEQUAL          ; 串相等
```

能转移到相应的代码段。

另一种可能是，两个元素比较时Z标志清零，说明两个元素不相同。差值的详细结果（SI中的字节比DI中的大还是小）保存在标志寄存器中，如同CMP操作的结果。通过用JB，JA，JAE等检查其他的标志，我们能够判断出究竟是源（SI）还是目的（DI）寄存器指向较小的串。主要的困难在于留在SI和DI寄存器的地址指向了错误的位置。具体地说，SI（DI）中的值已经越过了比较的串位置—或者说刚好在串结尾的下一个位置。

```
JB      SOURCESMALLER         ; SI指向的串小
; 如果到达此处，DI<SI
```

第三个有用的操作是从一个串中寻找某个值的出现（或者不出现）。（例如，一个包含浮点数的串将会有‘.’字符，否则，将是一个整数。）这由SACS?（SCAn串）指令实现。不同于先前的指令，这类指令只处理一个串，因此用一个索引寄存器（DI）。该指令将累加器（AL或AX，取决于数据的大小）中的值与串的每个元素进行比较，设置标志并且适时地更新DI。同样，如果使用REP?就会在CX为0到达串的结尾或者Z标志变为正确的值时停止，无论哪种情况最后的DI都是指向所感兴趣位置的邻近下一个位置。

使用REPZ前缀，我们能够跳过一个串开头和结尾处的空格。假设Astring包含一个100字节的数组，其中一些（在开头或结尾）是空格字符（ASCII值是32）。要找出第一个非空字符所在的位置，我们可以使用下面的代码段：

```
MOV     DI,Astring            ; 载入串
MOV     AL, 32                ; 向累加器载入空格字节
MOV     CX, 100               ; Astring串中有100个字符
CLD                           ; 清零方向标志
REPE    SCASB                 ; 扫描不匹配
JE      ALLBLANKS             ; 如果Z标志置位，没有发现不匹配
; 否则，DI指向了第一个非空格字符后的一个字节
DEC     DI                    ; 恢复DI，指向第一个非空格字符
```

要跳过尾部的空格，我们只要从末尾开始（即Astring+99）并设置方向标志以便从右向左进行串的操作：

```
MOV     DI,Astring            ; 载入串
ADD     DI, 99                ; 跳至串尾
MOV     AL, 32                ; 向累加器载入空格
MOV     CX, 100               ; Astring串中有100个字符
STD                           ; 方向标志置1
```

```
REPE   SCASB                  ; 扫描不匹配
JE     ALLBLANKS              ; 如果Z标志置位，没有发现不匹配
; 否则，DI指向了第一个非空格字符后的一个字节
INC    DI                     ; 恢复DI，指向第一个非空格字符
```

类似的前缀，REPNZ（或REPNE），只要Z标志未置位，也就是说只要元素不相同，指令就会反复执行。因此，要在串中找出第一个‘．’（ASCII值44），我们对第一段程序略做修改：

```
MOV    DI,Astring             ; 载入串
MOV    AL, 44                 ; 向累加器载入字节‘.’
MOV    CX, 100                ; Astring串中有100个字符
CLD                           ; 清零方向标志
REPNE  SCASB                  ; 扫描匹配
JNE    NOPERIOD               ; 如果Z标志清零，没有发现匹配
; 否则，DI指向了第一个‘.’字符后的一个字节
DEC    DI                     ; 恢复DI，指向第一个‘.’字符
```

最后一个常用的串原语就是将某个值反复地复制到一个串。例如，这可用于数组快速清零。STOS（存入串）将累加器的值复制到指定串。要将先前定义的空白数组存入全0，可以使用以下代码：

```
MOV    DI, Empty              ; 目标串
MOV    CX, 10                 ; 要存储10个元素
MOV    AX, 0                  ; 要存入的值是0
REP    STOSD
```

遗憾的是，这是8088对用户定义派生类型的唯一支持。例如，如果程序员想用多维数组，程序员就必须自己想好如何规划存储布局。类似地，一个结构或者记录只能用邻近存储单元表示，对于面向对象的编程没有提供支持。这必须通过高层的汇编器和/或编译器解决。

6.4.5 局部变量和信息隐藏

目前所定义的存储器组织有一个问题就是每个存储单元都隐含地定义用于整个计算机。采用高级语言编程常用的术语来讲，就是我们看到的每一个变量都是全局的——意味着能从程序中任何地方去访问或者修改（先前定义的）Greet数组。这也意味着整个程序只能有一个变量命名为Greet。比较好的编程习惯提倡使用局部变量，它具有私密、安全以及重用名称的能力。类似地，正如JVM中讨论的，只有跳转指令限制了程序员重用代码的能力。

6.4.6 系统栈

JVM和8088两者都支持子例程（子程序）。如同JVM的jsr指令，8088提供CALL指令与硬件栈配合。该指令将程序指针（IP）的当前值压入栈并且从转移参数所指的位置开始执行。对应的RET指令弹出栈顶值，将其装入指令指针，从保存的位置继续执行。

8088认定标准的PUSH和POP指令从机器栈移入/移出数据。例如，良好的编程习惯提倡不要破坏子程序中寄存器的内容，因为无法确定调用环境不再需要该数据。确保这不会发生的最简方式就是在子程序开始处保存（PUSH）这些寄存器并在结束的时候恢复（POP）它们。PUSH和POP指令均接受寄存器或存储器地址参数，PUSH AX和PUSH SomeLocn都是合法的。要压入一个常数值，应该首先将其载入存储器或者寄存器—当然，栈内容弹出到一个常数中就没有意义了。

多数汇编器不鼓励子程序调用和跳转语句使用同一个标签，尽管CPU并不关心这一点。（毕竟，这些标签只是加到程序计数器中的数值！）然而，如果对此没有认真处理，程序员将会违反栈规则，要么将多余的东西留在栈中（导致栈充满进而产生溢出错误），要么从一个空栈中弹出并使用垃圾内容。也就是说，不要这样去做。由于这一原因，8088汇编语言中的子

程序与我们已经见过的标签有些不同：

```
MyProc PROC
        PUSH    CX                          ; 将CX压入栈中保存
        ; 聪明地做些事情
        MOV     CX, 10
MyLabel:
        ; 在10次循环内做些聪明的事情
        LOOP    MyLabel
        POP     CX                          ; 恢复CX
        RET
MyProc ENDP
```

这里有几点需要注意。首先，为了帮助程序员和汇编器跟踪差异，标签MyProc的声明有别于标签MyLabel（例如，没有冒号）。第二，例程用PROC/ENDP开头和结尾。这些不是助记符，只是汇编制导，它们并不会翻译为机器指令。它们只是帮助程序员和汇编器组织程序。最后一条机器指令是RET，这是子程序需要的。第三，CX寄存器用于子程序内部循环的索引，因为值被压入栈顶并且在子程序结尾处弹出，所以调用环境看不到CX中的任何变化。从其他例程调用这个例程是合法的，调用如下：

```
Other   PROC
        MOV     CX, 50                  ; 调用MyProc 50次
LoopTop:
        CALL    MyProc                  ; 执行MyProc子例程
        LOOP    LoopTop                 ; 在循环中（50次）
        RET
Other   ENDP
```

Other过程使用同样的循环结构，包括CX，调用MyProc 50次，但是由于CX寄存器被保护了，所以不会出错误。当然，Other过程自身使用CX寄存器，所以调用Other必须注意。（更好的Other版本—专业程序员期待的—会在使用前先利用PUSH/POP指令对其做类似的保护。）

6.4.7 栈帧

除了提供临时存储的寄存器，存储器中的栈也可以用于临时存储局部变量。要理解这是如何工作的，我们首先看栈本身怎样工作（图6-5）。

8088的一个通用寄存器SP保留给CPU和操作系统用来保存机器栈顶当前的位置。在任何程序的开始，SP的值被设置为靠近主存顶端的某个位置—主存相对高地址的部分，而程序本身被保存在较低的位置。在程序和栈顶之间是大片的未用空闲存储区。

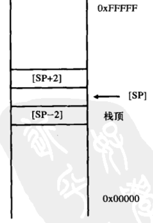
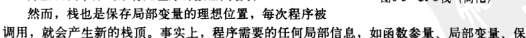


图6-5 CPU栈（简化）

一旦数据被放入栈中（一般由PUSH或者CALL），SP寄存器就减去一个适当的量，通常是2。这将栈顶又向程序部分推进了2个字节。压入的值就保存在这2个字节中。另一个PUSH会使SP再降2个字节并保存另外的数据。与直觉正好相反，这意味着栈“顶”实际上是栈的最低（最小）地址部分。当数据从栈弹出时，[SP]处的值被复制到变量中；然后SP加2，设置为新的“栈顶”。（这也适于执行RET并且从栈中弹出原有的程序计数器内容。）

然而，栈也是保存局部变量的理想位置，每次程序被调用，就会产生新的栈顶。事实上，程序需要的任何局部信息，如函数参量、局部变量、保



存的寄存器内容以及数据都能放入栈内。既然这是一个常见任务，特别是在高级语言中，就有一个标准方式组织栈以便不同复杂程度的程序能够协调运行。

这一基本思想称为栈帧（图6-6）。正如通常所实现的，它包含两个寄存器，SP和BP。这就是为什么程序员不应该乱用BP寄存器进行一般索引的原因；因为BP已经用于栈帧。但是因为BP作为索引寄存器，所以MOV AX, [BP+4]这类的表示是合法的并且能用来访问靠近BP的存储单元。

有了以上认识，栈帧如下（从存储器顶部开始，或者从栈"底"开始）：

- 过程子程序的任何参量
- 返回地址
- BP原有的值（由BP指向）
- 保存局部变量的空间（由SP指示顶端）
- 保存的寄存器内容

图6-6　一个8086栈帧

这是如何工作的？我们下面使用一个有些复杂的例子：我想要编写一个等价于further()函数或方法的汇编语言代码，further()采用了两个参量并且返回了一个大于零的绝对值。用Java编写的这个方法如图6-7所示；用C/C++编写的如图6-8所示。

```
public static int further(int x, int y)
{
        int i,j;
        i = x;
        if (i < 0)
                i = -x;
        j = y;
        if (y < 0)
                j = -y;
        if (i < j)
                i = j;
        return i;
}
```

图6-7　Java函数further(int x, int y)

```
int further(int x, int y)
{
        int i,j;
        i = x;
        if (x < 0)
                i = -x;
        j = y;
        if (y < 0)
                j = -y;
        if (i < j)
                i = j;
        return i;
}
```

图6-8　C/C++函数further(int x, int y)

代码本身很简单；假设我们将数据移入AX和BX，就能简单地比较这两个数，如果BX大，就将其移入AX。类似地，将函数的参数（x和y）与0比较，我们可以对局部变量使用NEG指令也可以不用NEG。然而，要正确地访问它们，我们需要足够的栈空间保存这些局部变量。

下面的代码将很好地解决这个问题：

```
Further     PROC
      PUSH  BP              ; 保存原有BP
      MOV   BP, SP          ; BP现在指向SP内容
      SUB   SP, 4           ; 留出两个16位局部变量空间

      PUSH  BX              ; 保存BX

      MOV   AX,[BP+2]       ; 获取第一个参量
      MOV   [SP+2], AX      ; 保存在第一个局部变量位置
      CMP   [SP+2],0        ; 与0相比
      JGE   Skip1           ; 如果>=0就不求反
```

```
        NEG     [SP+2]              ; 否则，求反
Skip1:

        MOV     AX,[BP+4]           ; 获取第二个参量
        MOV     [SP+4], AX          ; 保存在第二个局部变量位置
        CMP     [SP+4], 0           ; 与0相比
        JGE     Skip2               ; 如果>=0就不求反
        NEG     [SP+4]              ; 否则，求反
Skip2:
        MOV     AX, [SP+2]          ; 载入第一个局部变量
        MOV     BX, [SP+4]          ; 载入第二个局部变量
        CMP     AX, BX              ; 如果第一个局部变量>=第二个局部变量
        JGE     Skip3               ; 那么不做调整
        MOV     BX, AX              ; 否则，求反
Skip3:                              ; 答案现在AX中

        POP     BX                  ; 恢复原来BX的值
        ADD     SP, 4               ; 销毁局部变量
        POP     BP                  ; 恢复原来BP的值
        RET                         ; 仍然需要弹出参量
Further ENDP
```

图6-9展示了这个程序构建的栈帧。特别注意传入的两个参量以及原有寄存器值被保存的位置。最后，有两个存储器字与局部变量i和j对应。在程序结尾处栈最终以相反的次序解构。为什么我们不保存并恢复AX中的值？因为AX寄存器被用于保持返回值，因此它的原有内容不得不被覆盖。

这是如何工作的？我们将使用一个有些复杂的例子：我想编写一个与函数further(−100，50)（它将返回100）等价的汇编语言程序。为了调用这个程序，调用函数必须首先将两个整数压入栈；然后还必须找到一种方法在函数返回后撤销这两个值。完成这个任务的简易方法如以下代码所示：

```
X       DW      -100        ; 第一个变量
Y       DW      50          ; 第二个变量
PUSH    X                   ; 压入X
PUSH    Y                   ; 压入Y
CALL    FURTHER
 ; 此时AX包含值100
ADD     SP, 4               ; 从栈中移出X, Y
 ; AX仍然包含值100
```

所构建的整个栈帧如图6-9所示。

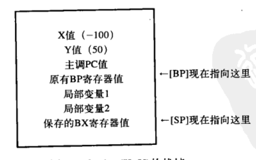


图6-9 further(X, Y)的栈帧

6.5 再论锥形山

作为一个展现8088如何进行算术运算的有效示例，让我们来分析先前给出的山体体积问

题。如第2章所述，原问题描述如下：

底部呈圆形、直径450m、并且高150m的山体体积是多少？

因为答案中涉及使用π，所以至少计算部分应该在浮点运算单元FPU中完成。我们假设（为了清楚起见）名称/标识符"STORAGE"指示一个16位的存储单元（与6.4.1节中一样；这可以是某处的全局变量或者是栈中的局部变量），并且我们可以使用这个单元将数据移入/移出FPU。为了简单起见，我们使用16位整数以及寄存器进行计算。假设（为了清楚起见）完成整型计算；因为涉及的数不是很大，这不会出现问题。

首先，我们计算半径：

```
MOV    AX, 450              ; 直径=450m
MOV    BX, 2                ; can't divide by a constant
DIV    BX                   ; AX=450/2, DX仍为0
```

底部的面积是半径的平方乘以π，为了使用π，我们必须初始化FPU，并且在此进行运算。

```
                            ; AX中保存半径
MUL    AX                   ; AX平方，结果为DX：AX
FINIT                       ; 初始化FPU
MOV    STORAGE, AX          ; 将半径的平方（作为整数）移入存储器
FILD   STORAGE              ; 移入FPU
FLDPI                       ; 载入pi=3.1415...
FMUL                        ; 计算底的面积
```

此时，我们可以将底的面积从FPU移至ALU，但是这样做之后，我们会丢掉小数点右面的部分（即损失精度）。更好的方案就是继续在FPU中计算，用存储单元STORAGE作为整型数据的临时单元。重申要点：圆锥体积是相应圆柱体积的三分之一，而圆柱体积等于底的面积（已经算出）乘以高（150m）。因此，最后的计算过程如下：

```
MOV    STORAGE, 150         ; 将高度载入FPU
FILD   STORAGE
FMUL                        ; 圆柱体积=底的面积×高
MOV    STORAGE, 3           ; 将3移入FPU用于除法
FILD   STORAGE
FDIV                        ; 除以3，求圆锥体积
; 结果现在FPU栈顶
FISTP  STORAGE              ; 'P' 意味着运算执行后弹出栈（保存）
; STORAGE现在包含最终结果，舍入为整数
FWAIT                       ; 在ALU进行操作之前等待FPU完成
```

6.6 接口问题

前面例中的代码是汇编语言程序与外界接口的一种方式；这些栈帧也允许高级语言生成的代码（有些微小的变化，需注意）来操纵。因此，如果你有一个大型的Pascal或者C++程序，可以用汇编语言编写一小部分（一个或几个函数）以确定你可以完全控制整个机器——例如获取游戏动画效果的额外的加速。

尽管多数人考虑到接口时通常认为是与设备或者外设的接口。例如，我如何将来自键盘（用户从这里键入）的数据送至CPU，然后送至屏幕（数据在此显示）？可惜答案是："这要看具体情况"。

事实上，这取决于很多事情，你使用的设备类型，你所用的机器种类，以及你正在运行的操作系统的类型。实际上，任何操作系统（Windows，Linux，MacOS，FreeBSD，等）都

是一种特殊的计算机程序，它始终都在运行并且介入在系统上的其他程序与硬件设备之间。操作系统提供并且控制对输入输出设备的访问——这意味着如果你想要同设备打交道，必须调用相应的函数（由操作系统提供），并借助于在栈中设置正确的参量以及在合适的位置发出CALL。这些函数的细节因系统而有不同。Linux系统采用一种方式工作，使用一组函数。Microsoft Windows做同样的事情却使用一组不同函数，它需要不同的参量及调用。因此，要与多数标准设备交互，秘密在于理解你的操作系统如何处理它，然后使用与操作系统的神奇“握手”，让操作系统去完成你想要做的事。

有其他两个主要方法与设备接口。一些设备，如视频控制器，可以直接归属到计算机存储器上并且当相应的存储内容改变时就自动地更新。显然这个存储区对于其他的用途（比如存储程序）是不可用的。在最初的PC上（运行MS-DOS），例如，“视频存储器”（VRAM）起始于0xA0000，这意味着程序不能使用超出0x9FFFF的空间。然而，这也意味着一个聪明的程序通过将合适的数据放在合适的超出0xA0000的位置上就能在屏幕上显示该数据。这个技术，称为存储器I/O映射，可以很容易地实现，例如，将ES段寄存器设置为0xA000，然后使用寄存器对如ES：AX而不用更一般的DS：AX作为MOV指令的目标变量。

其他设备通信方式是通过不同的端口（如串口，UDP端口等）。这通常称为端口I/O映射。每个这样的端口（以及其内部不同的数值，如视频色彩）都可用一个16位的端口标识号进行独立寻址。OUT指令有两个参量，一个16位端口和一个8位数据值，仅将数据值传到端口就可将该数据送至附加在该端口的设备中。设备对该数据的处理是它自己的事。IN指令将从指定的端口读出一个字节的数据。显然，这种类型的编程需要非常了解端口编号体制以及数据类型和含义。但是有了这种控制，如果你要用，你就可以将你的壁挂鱼缸钩到8088的打印端口，不是用于打印，而是自动控制鱼缸的温度和换气。

6.7 本章回顾

- Intel 8088是系列芯片的先驱。特别是，作为最初IBM-PC内部的芯片，8088很快就成为商用桌面机的最常用芯片，并且确立了IBM和Microsoft作为20世纪大部分期间的工业霸主地位。
- 8088是CISC芯片设计的经典案例，CISC芯片拥有庞大、丰富的指令集。
- 8088有8个16位的寄存器，被命名为“通用”寄存器（尽管这些寄存器中许多都有特殊的用途），以及一些小的8位寄存器，这些8位寄存器在物理上是16位寄存器的一部分，还有逻辑上（以及物理上）分开的浮点单元FPU
- 大多数汇编语言操作采用两变量形式，操作助记符后面跟目的变量和源变量，如下：

```
OP      dest,src             ; dest = dest OP src
```

- 可用的操作包括通常的算术运算（尽管乘法和除法有特殊的格式并且使用特殊寄存器）、数据传送、逻辑运算，以及若干其他特殊用途的操作功能。
- 8088支持多种寻址方式，包括立即方式、直接方式、间接方式和变址方式。
- 浮点操作在FPU中进行，使用基于栈的标志和一组特殊的操作（大部分以字母F开头）。
- 8088支持标准的分支转移指令，也支持特殊的循环指令，使用CX寄存器作为循环计数器。
- 8088在存储器中保存数据的格式与在寄存器中保存数据的格式不同，这会使编程新手感到困惑。
- 数组和串使用邻接存储单元实现，也有特殊用途的串原语操作用于常用的串/数组操作。
- SP和BP寄存器通常用来支持具有标准栈帧的标准化机器栈；这使得汇编语言代码与高级语言代码的结合很容易。

6.8 习题

1. “系列芯片”的含义是什么？
2. 为什么8088有固定数量的寄存器？
3. BX和BL寄存器之间的差别是什么？
4. 下面的段:偏移所对应的实际地址是什么？
   a. 0000:0000
   b. ABCD:0000
   c. A000:BCD0
   d. ABCD:1234
5. 在JVM中没有的CISC指令的实例？
6. 乘法指令和加法指令有何区别？
7. JA和JG之间的区别是什么？
8. ADD WORD PTR [4000h]，1和ADD BYTE PTR [4000h]，1之间有何不同？
9. 8088在使用16位UNICODE量表示字符的语言（如Java）中如何处理串操作？
10. 8088程序中所谓的全局变量是如何保存？
11. 子程序的参数以自左至右还是自右至左的次序进栈对于8088有影响吗？



第7章 Power体系结构

7.1 背景

对基于Intel芯片设计的用于台式机的CPU而言，唯一最大的竞争者可能就是Power体系结构。直到2005年中期，PowerPC芯片一直用在Apple Macintosh计算机中。尽管Apple已经转向了基于Intel的芯片，Power体系结构仍然在许多应用领域中起主导作用。截至2005年11月，世界上20个最快的超级计算机中半数使用基于Power的芯片，三个主要的游戏平台（Sony，Microsoft和Nintendo）到2006年底全都计划采用Power芯片实现控制台。在嵌入式系统中，2006个汽车模型中约有一半使用基于Power的微控制器（由Freestyle生产）。

如果说Pentium是复杂指令计算CISC体系结构的权威示例，Power芯片就是精简指令集计算RISC体系结构的教科书版本。

从历史的角度，Power体系结构创始于1991年Apple、IBM和Motorola的联合设计项目。（注意Intel不是这个联盟的成员。要知道Intel由于基于CISC的x86系列已经占据了统治市场地位，因此何必参加这个联盟？）RISC由IBM一直在嵌入式系统（见第9章）中使用了近20年，被认为是一种从相对较小的（因此廉价的）芯片获取高性能的方式。

RISC计算的关键在于（如同生活中的很多事）计算机程序花费大部分时间去做一些相对常用的操作。例如，研究表明在一个典型的程序中大约20%的指令就是load/store指令，即数据在主存和CPU之间的移动。如果工程师能够加倍这些指令的操作速度，那么他们就能获得系统整体性能10%的改进！所以，不是花时间和精力去设计执行复杂任务的硬件，他们的任务是设计执行简单任务的硬件，以使任务执行的性能好、速度快。另一个极端是，增加不常使用的寻址方式会降低每条指令的执行性能，因为计算机必须检查每一条指令来看该指令是否使用了这种寻址方式，这需要更多的时间或者昂贵的电路系统。

一个典型的RISC芯片体系结构通常有两个特别的方面可加速（并简化）计算。第一，指令本身通常是固定长度（对于PowerPC，所有指令都是4个字节长度；对于Pentium，指令长度可变，在1~15字节之间）。这对于CPU取指令部分可以完成得更容易并且更快，因为它不需要花时间考虑究竟取多少字节。类似地，二进制模式的译码以决定执行什么操作也能完成得更快并且更简单。最终，每条指令都足够的简单从而可以快速地执行（通常在它取下一条指令的时间里执行完成——单周期）。

RISC体系结构的第二个优点与指令组有关。它不仅能很快地执行每一条指令，而且指令集中的指令数量少，这意味着在如何进行给定的（高级）计算问题上通常没有大量的近义变形。这使得优化的编译器能很容易地分析代码。更好的分析可以产生更快的程序，即便单个的指令并没有加速。

当然，RISC体系结构如Power不足的一面就是大部分功能强大的操作（例如，第6章描述的串处理系统）不作为单独的指令存在，必须用软件来实现。甚至许多存储器访问指令和方式也不可用。

Power联盟的设计目标之一就是不仅开发有用的芯片，也要开发交叉兼容的芯片。当然，

Motorola和IBM已经建立了良好的独自设计的芯片系列（例如，IBM的RS/6000芯片以及Motorola 68000系列）。通过事先达成某种原则，Power联盟能确保他们的芯片可彼此互操作，对于未来技术的改进也提前进行了设计。如同80x86/Pentium系列，Power是芯片系列（包括PowerPC子系列）——但是与80x86/Pentium不同的是，Power设计提前考虑了可扩展性。

Power对灵活性方面的关注已经带来了奇特的效果。首先，与Intel系列不同，开发的焦点不是生产新的、更大的机器。从一开始，低端桌面甚至嵌入式系统控制器就已经是目标市场的一部分了。（这一市场优势应该是明显的：如果你的桌面工作站与你的烤箱制造厂所用的控制芯片100%兼容，编写和调试该烤箱的控制软件就很容易。因此，IBM不仅能卖出更多烤箱控制器芯片，还能卖出更多工作站。）与此同时，该联盟计划最终扩展到64位领域（现在典型代表是PowerPC G5）并且定义一个最初（32位）设计的扩展指令集来处理64位的量。此外，PowerPC在数据存储格式方面提供了大量变形，这在下面将会看到。

7.2 组织和体系结构

正如大多数现代计算机，为了支持多任务，PowerPC系统至少有两个不同的视角（形式上称为编程模型，也常称作编程模式）。其基本思想就是用户级程序只具备有限的权限能访问计算机的一部分资源，而相当一部分计算机资源超出了用户级程序的访问权限，除非程序（典型的是操作系统）运行在超级用户的特权下。（8088的不安全性就是由于缺少这样明确的编程模型。）这可防止用户程序彼此干涉或者妨碍关键的系统资源。本章的讨论将主要关注用户模型，因为这是与日常编程任务最相关的。

7.2.1 中央处理单元

通过图7-1可以了解Power体系结构CPU的大部分特征。有一组“通用寄存器”，数量为32个（相对较多），在早期的PowerPC型号如601（直到G4）中是32位宽，而在G5和970中是64位宽。外部还有32个浮点寄存器，其宽为64位。ALU和FPU两者都有状态/控制寄存器（CR和FPSCR），并且有几个（相对较少）芯片特有的专用寄存器，你可能不需要用它们来保持PowerPC系列之间的兼容性。或者至少这就是要告诉给用户的。

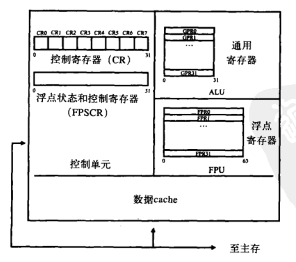


图7-1 PowerPC CPU体系结构方框图

实际的物理布局更加复杂一些，因为有些专用硬件只有超级用户（也就是操作系统）可以使用。例如，有一个机器状态寄存器（MSR）保存系统方面的超级用户级别的重要信息。（这类信息的一个例子是系统当前运行在用户级还是超级用户级，也许是由外部设备产生的响应超级用户级事件。另外一个系统方面信息的例子是存储器存储格式是采用大端（big-endian）还是小端（little-endian）格式，后面会讨论。）将当前的计算状态（保存在CR和FPSCR中）与整个机器状态分开可以使多任务环境对于突发变化的响应更容易一些，不会中断当前的计算。一个更加微妙的优点就是CPU芯片设计者能够复制CR和FPSCR并允许几个不同的用户级程序同时运行，每个程序使用它自己的寄存器组。这个过程在理论上可以应用到整个用户级寄存器空间。特别地，PowerPC G5复制了整型和浮点处理硬件，允许来自于最多4个进程的多达4条指令完全并行执行。

用户与超级用户对CPU芯片视图的另一个关键区别是关于存储器访问。操作系统必须能访问（并且真正控制）计算机全部可用的存储器，出于安全的理由用户级程序通常被限制在它自己的空间里。PowerPC有一组固定的寄存器用于在硬件级别上管理存储器访问，但是只有超级用户能够访问这些寄存器。下一节将描述这个功能。

7.2.2 存储器

存储管理

在用户级，PowerPC存储器组织简单（比8088简单得多）。每个用户程序都拥有32位地址空间，在理论上可访问2^{32}个不同的一维存储单元。（当然，对于64位版本的PowerPC存在2^{64}个不同的地址/单元。）这些单元定义了程序可用的逻辑存储器。它们要么用作直接地址（如先前所定义）要么用作逻辑地址，逻辑地址由存储管理硬件进行转换。此外，PowerPC定义了第3种用来快速访问指定的存储区域的访问方法。

块地址转换

如果一个特别的存储器块需要经常（而且快速地）被访问，CPU有一组专用寄存器，块地址转换（BAT）寄存器，用来定义物理存储器的特殊块，可以用于执行类似的查找任务但只要很少的步骤。BAT寄存器最常用于代表高速数据传输的存储区域，例如图形设备和其他类似的I/O设备。如果一个逻辑地址与BAT寄存器标记的存储区对应，那么就跳过虚拟存储过程，进而从BAT寄存器直接读出对应的物理地址。

cache访问

存储器访问的最后阶段与如何完成访问无关，而是要决定所需的内存位置之前（近来）是否已经访问过，或者更精确地，所需的数据是否保存在CPU内部的cache中。因为片内存储器的访问要比片外主存的访问快很多，如同大多数现代处理器芯片，PowerPC将近期使用的数据拷贝在一个很小但速度很快的存储器中。

7.2.3 设备和外设

先前定义的存储管理系统的另一特征就是访问I/O设备能在存储管理系统中容易地处理。利用BAT寄存器访问高速I/O设备已经提到过。类似的方法（I/O控制器接口转换）能够完成虚拟存储系统内类似的存储器地址任务。每个段寄存器包含信息以及VSID。这里的信息域详细指出该逻辑地址是否指向某个外设。如果是，就跳过页转换，并且该逻辑地址用来产生指令和地址序列来处理适当的设备。这使得PowerPC对于设备访问和存储器访问同样容易（实际上就像存储器访问），只要操作系统在段寄存器中正确地设置了数值。

7.3　汇编语言

7.3.1　算术运算

PowerPC汇编语言与我们已经看到的系统有两个明显的不同。第一，尽管PowerPC有一个寄存器组（如同x86和Pentium），但是寄存器采用编号而不是名字。第二，Power体系结构指令拥有独特的3参量格式。将两个数相加的指令可写为：

```
add r3,r2,r1     # 寄存器3 =寄存器2 +寄存器1
```

注意第2和第3个参量（寄存器2和1）在计算中保持不变（这有别于我们已学过的其他CPU），因此可以在后来被重用。当然，通过参量的重复，我们能得到更传统的操作，例如：

```
add r2,r2,r1     # 等价于x86的ADD R2, R1
```

一般地，二进制操作（算术，逻辑，甚至比较）会以这种方式表示。这种3参量格式，结合相对较多的可用寄存器，在进行计算和保存中间结果方面给程序员和编译器带来了巨大的灵活性。它也给予计算机一定的余地重新按需动态地组织计算。如果你考虑下面的指令序列

```
add r3,r2,r1     # (1) 寄存器3 =寄存器2 +寄存器1
add r6,r5,r4     # (2) 与上面的类似
add r9,r8,r7     # (3)
add r12,r11,r10     # (4)
add r15,r14,r13     # (5)
add r18,r17,r16     # (6)
```

没有理由要求计算机一定按上述次序执行，有足够能力的计算机甚至能够同时执行这些指令。

一个更加有趣的问题是为什么Power体系结构以这种3参量的方式工作。一个简单的回答是，“因为它能够”。在设计上，PowerPC对每条指令都采用统一的并且相当大的位数——32位。由于有32个寄存器，指令用5位指定任一给定的寄存器，所以指明3个寄存器就要占用32位中的15位，这还意味着可以有2^{17}（大约128000）个不同的3参量指令可用。显然，这已经超出任何神志正常的工程师的设计期待——但工程师们不是让add指令比其他的指令短些，而是在提供额外的灵活性方面为特大空间找到了用途。

然而，有一个领域没有提供额外的灵活性，这就是存储器寻址。一般地，PowerPC的算术和逻辑操作不能访问存储器。正式地说，只有两种寻址方式，寄存器方式和立即方式，如前面章节所定义。不同方式由特殊的操作码和助记符区分；一个无标记的助记符针对三个寄存器，而以-i结尾的助记符对两个寄存器和一个16位立即值进行操作，例如：

```
add r3,r2,r1     # 寄存器3 =寄存器2 +寄存器1
addi r6,r5,4     # 寄存器6 =寄存器5 +4
```

（在一些汇编器中，这里可能存在一些困惑，因为允许程序员用编号引用寄存器而不用r?标志。语句add　2，2，1会将寄存器1的内容加到寄存器2中；它不是加数字1。要做得更好些，编写代码时就不要粗心，即使汇编器允许你这么做。）

当然，只有一个16位立即值可用（为什么？），我们不能对一个32位寄存器的高半部分进行操作。有些操作（add，and，or和xor）采用了-is作后缀的变形（这些指令对立即值进行左移16位的操作，允许程序员直接影响寄存器的高位）。所以语句

```
andis r3,r3,FFFF     # r3 = r3 and 0xFFFF0000
```

将用32位模式0xFFFF0000与r3的内容逻辑与，因此将寄存器的低半部分置为0。类似地，

```
andis r3,r3,0000     # r3 = r3 and 0x0000000
```

将整个寄存器清零。当然，也可以用寄存器-寄存器操作获得如下相同的效果：

```
xor r3,r3,r3      # 用r3与其自身异或（清零）
subf r3,r3,r3     # r3减去其自身（清零）
```

（即使在RISC芯片上，也有多种方式来完成同样的功能。）

你所期待的大部分算术和逻辑操作都具备了，这些操作有加（add，addi）、减（subf，subfi）、求反（算术反转，neg）、与（and，andi）、或（or，ori）、异或（xor）、与非（nand）、或非（nor）、乘和除等。还有一组功能强大的移位和循环移位指令。为了简化芯片的设计逻辑，不是所有的这些操作都有立即形式（毕竟是精简指令集计算），但是很少会有程序员失去立即方式nand指令的便利。一些汇编器提供了助记符别名——例如，not指令并不是由PowerPC CPU提供的。它能用来模拟“同常数0的nor”，因此不是必须由PowerPC提供。反而一个聪明的汇编器会辨识not指令并且输出等价的结果而没有异议。像通常一样，乘和除有些复杂。经典的问题就是两个32位因子相乘通常会产生64位的乘积，它无法装入一个32位的寄存器中。因此，PowerPC支持两个独立的指令mullw和mulhw，它们分别返回乘积的低字和高字。这些操作都使用通常的3参量格式，所以

```
mullw   r8,r7,r6
```

计算乘积r7 · r6然后将低字放入r8。除法通常分为有符号除和无符号除：divw和divwu。这些助记符中‘w’的含义表示对字（word）的操作。

7.3.2 浮点操作

对于浮点数的算术指令类似，但是要用64位浮点寄存器而不是普通的寄存器，并且助记符以一个f开头。因此两个浮点数相加就是fadd。默认情况下，所有浮点指令都对双精度（64位）量进行操作，但是内部有一个s的变形（如，faddsx）指定32位值。更进一步，FPU能够以整数格式存储/载入数据，FPU自身能够处理浮点数和整数之间的转换。

在PowerPC上编写代码的一个危险是不同寄存器共享相同的编号。例如，指令

```
fmul  r7, r8, r9      # 乘
add   r7, r8, r9      # 加
```

两条指令好像对同一组寄存器进行操作。实际上不是。第一条语句对浮点寄存器操作，而第二条是对通用寄存器操作。在一些汇编器中，在恰当的场合下，自然数也被解释为寄存器编号。这意味着语句

```
add   7, 8, 9         # 加
addi  7, 8, 9         # 加
```

完全不同。第一条将寄存器8和寄存器9中的值相加，而第二条将寄存器8中的值与立即整型常数9相加。特别警告。

7.3.3 比较和条件标志

大多数计算机在缺省情况下会在每个算术或逻辑操作之后更新条件寄存器相应的内容，PowerPC则稍有不同。第一，已经提到过，PowerPC没有单一的条件寄存器。更重要地，比较只在明确需求时进行（这有助于支持为了获取速度重排指令次序的思想）。最简单的请求比较方式就是在整型操作码末尾添加句点符（.）。这将根据运算结果大于、等于或者小于零来设置条件寄存器CR0的位段。

更一般地，两个寄存器（或者寄存器与立即方式的常数）之间的比较使用cmp指令的变形。这可认为是最奇特的3参量指令，因为它不仅带有两个要比较的参量（第二和第三参量），还有条件寄存器的索引。例如：

```
cmpw     CR1, r4, r5       # 比较r4和r5
                           # 结果在CR1中
```

设置4位条件寄存器1中的位来反映r4和r5的关系，如表7-1所示。

表7-1 cmpw CR1, r4, r5执行后CR1中的值

	该位设置为1的含义
bit0	r4 < r5（结果小于0）
bit1	r4 = r5（结果等于0）
bit2	r4 > r5（结果大于0）

7.3.4 数据移动

如同JVM，要将数据移入/移出存储器，可用专门的载入和存储指令。载入指令一般有两个参量，第一个参量是目的寄存器，第二个参量是要载入数据的逻辑地址（也许要经过存储管理，前面已描述）。和JVM一样，有不同的指令对应不同大小的数据移动；载入数据的指令都以字母l开头，接下来的字符指明要移动的数据是一个字节（b）、半字（h，2个字节）、字（w，4个字节），还是双字（d，8个字节；明显地只在PowerPC的64位版本上可用，因为只有这个版本能够在寄存器内存储如此大的量）。载入指令显然也能加载单精度（fs）和双精度（fd）浮点数。当然，不是所有的数据都从存储器载入。li指令用立即方式载入常数。

当载入量少于一个字时，有两种不同的方式填充寄存器的其余部分。如果指令指明“零载入”（使用字母z），寄存器的高位部分就被设置为0，而如果指令指明“代数载入”（a），即对寄存器的高位部分进行符号扩展。最后，有一个更新模式可用，用u指定，将在下面解释。

要了解下面的实例，假设（EA），“有效地址”的简写，是一个存储单元，该单元内存有适当容量的存储块，目前保存的是全1位模式。指令lwz r1,（EA）将位于（EA）的字载入到寄存器1中并且（在64位机器）将寄存器其余部分置0。在一个32位PowerPC上，寄存器1将包含值0xFFFFFFFF，而在64位机器上，寄存器1将保存值0x00000000FFFFFFFF。表7-2给出了载入指令的其他一些工作示例。

表7-2 PowerPC载入指令的一些示例

指令	32位寄存器结果	64位寄存器结果
lbz r1,(EA)	0x000000FF	0x00000000000000FF
lhz r1,(EA)	0x0000FFFF	0x000000000000FFFF
lha r1,(EA)	0x FFFFFFFF	0xFFFFFFFFFFFFFFFF
lwz r1,(EA)	0x FFFFFFFF	0x00000000FFFFFFFF
ld r1,(EA)	不允许	0xFFFFFFFFFFFFFFFF

在此有一些警告。也许最明显的是，即使在一个64位机器上，也无法“扩展”一个64位双字量。在32位PowerPC上也没有办法直接对双字量进行操作，或者32位CPU中没有办法将字进行代数扩展至双字。因此，32位PowerPC指令集中不含lwa指令，并且lwz指令载入一个32位量但是不做任何零扩展。最令人困扰的，PowerPC体系结构不允许对字节量进行代数扩展，所以指令lba也不存在。除了这些例外，载入指令非常完备并且很好理解。例如，指令lsu r3,（EA)从存储器载入了一个单精度浮点数，使用了迄今为止还未定义的“更新”和“索引”模式。

从寄存器向存储器存储数据的操作类似，但是以“st”开头并且不用考虑扩展问题。指令

sth r6，(EA)将寄存器r6的低16位保存在EA存储单元，而sthu，sthx或者sthux指令完成同样的功能，但是分别使用更新模式，索引模式或两者结合。

7.3.5 转移

汇编语言程序设计最后一个基础方面就是控制/转移指令。正如我们所期待的，PowerPC支持无条件和条件转移。无条件转移的助记符就是一个简单的b，带有单独一个目标地址。条件转移指令（有几种变形，统称为b？）包括一个参量和条件寄存器，如下：

```
bgt   CR0, address      # 如果CR0的位1被置1则转移
blt   CR1, address      # 如果CR1的位0被置1则转移
beq   CR2, address      # 如果CR2的位2被置1则转移
ble   CR3, address      # 如果CR3的位1被置0则转移
bne   CR4, address      # 如果CR4的位2被置0则转移
bge   CR5, address      # 如果CR5的位0被置0则转移
```

PowerPC不支持单独的跳转–至–子程序指令，但是有一个专用链接寄存器完成类似的任务。还有一个用于循环的专用计数寄存器。这两个寄存器都由专用的指令使用。

尽管RISC设计简洁，仍然有很多可用的指令，我们不能一一介绍。特别是，超级用户级寄存器如BAT和段寄存器都有自己的指令进行访问。如果需要为Power芯片编写操作系统，那么你首先需要读些其他的书。

7.4 再论锥形山

再来看我们的老朋友，锥形山，这是我们已经举过的例子。为了表示的简洁性，这个例子假设可用64位操作。同样为简单起见，假设π值在存储器中可用，表示为存储单元(PI)。如第2章所述，原问题描述如下：

底部呈圆形直径450m并且高150m的山体体积是多少?

由于可用寄存器很多，解决该问题就十分简单了：

第一步是计算半径，用直径除以2，然后再求其平方。

```
li    r3, 450          # 直径载入r3
li    r4, 2            # 除以2得到半径
divw  r3, r3, r4       # 将半径放入r3
mullw r3, r3, r3       # 半径平方
```

第二步将数据（经由存储器）移至FPU。

```
std   (EA),r3          # 将半径平方存为64位整数
ldf   r10, (EA)        # 载入半径平方
fcfid r9, r10          # 转换为浮点
```

第三步载入π，并与其相乘。

```
ldf   r8, (PI)         # 载入用于乘法的π
fmul  r7, r8, r9       # 相乘
fcfid r9, r10          # 转换为浮点
```

最后，载入高度（150）并作乘法，然后将最终量除以3。为了清除起见，我们首先以整数形式加载，然后将它们传给浮点处理器，如下：

```
li    r5, 150          # 载入高度
std   (EA),r5          # 存为整数
ldf   r6, (EA)         # 将高度载入FPU
fcfid r6, r6           # 转换为浮点
fmul  r7, r6, r7       # 相乘

li    r5, 3            # 载入常数3
std   (EA),r5          # 存为整数
```

```
ldf   r6, (EA)           # 将3载入FPU
fcfid r6, r6             # 转换为浮点
fmul  r6, r6, r7         # 相乘
```

注意在这个例子中，寄存器3到寄存器5总是用于保存整数，因此一直是通用寄存器，而寄存器6到寄存器10总是用于保存浮点数。这只是为了保证解释的清晰。

7.5　存储器组织和使用

一旦你了解了超级用户级存储器管理，PowerPC存储器组织就简单了。如前面所讨论的，从用户的角度，PowerPC提供了一个简单扁平的存储空间。存储空间的容量是CPU字长的一个简单函数——对于32位CPU是2^{32}字节（大约4GB）而对于64位CPU是2^{64}字节。当然，没有计算机能拥有16EB的物理存储器，但是这个容量为未来突破存储器成本留出了空间。存储器地址就是适当容量的编号：半字，字，和双字在存储空间中按适当对齐的间隔得到保存。（实际上，这是一个谎言。字节就是字节。但是指令例如`ld r0, (EA)`只有当(EA)指向一个8的倍数的有效地址时才能工作。否则，一个智能的汇编器/编译器就需要将双字载入拆分成8个部分载入和移位指令，这会减慢代码的速度。所以，不要采用这种做法。佯装对象已经按照合适的对齐位置保存，这样会感觉更好。）

PowerPC的一个关键特征是直接支持嵌入到CPU指令集的big-endian和little-endian数据存储。这个特征继承自IBM和Motorola，它们都有大规模的产品系列和大量的代码需要支持，但是在这方面做出了不同的选择。CPU中的数据总是以相同的方式保存，但是在存储器中数据要么以正常格式要么以“字节反向”格式存储。

除了这些复杂性之外，存储器寻址操作相当简单。PowerPC支持两种基本寻址方式，间接寻址和索引寻址；唯一的区别在于涉及的寄存器数量。

在间接方式中，计算机用16位立即偏移值与单一寄存器指定的内容之和作为有效地址。例如，如果寄存器3中有值0x5678，那么指令

```
lwz      r4, 0x1000(r3)
```

将保存在0x6678（0x1000+0x5678）单元的值载入到寄存器4的低32位（w）。在64位的机器中，因为z所以高32位被设置为0。对于大多数程序而言，0x1000由编译器定义，指向某个变量的偏移（如上所示）。

在索引方式中，有效地址的计算也类似，但是用两个寄存器代替一个常数和一个寄存器。所以，在下面的指令中

```
lwzx      r4, r2, r3
```

仅当寄存器2保存0x1000时有效地址是相同的。这提供了十分简单但是有用的两步方式访问存储器；第二个参量可用来定义一个特别的存储块——例如，全局程序变量的存储区——而第三个变量用来选择该块内的一个偏移（这在思想上与8088定义的段寄存器类似但更加灵活）。如果出于某种原因，该块作为一个整体被更换，只有一个寄存器需要改变，整个代码可以继续工作。

对于许多应用而言，特别是那些涉及数组的应用，访问存储器时能够改变寄存器值将是非常方便的（Java程序员已经熟悉了‘++’和‘--’操作符）。PowerPC通过更新方式提供这类功能，在助记符中用u表示。在更新方式中，有效地址的计算依旧，但是寄存器内的值被更新为有效地址。

举个例子，考虑下列语句的效果，这可能在一个循环的中间：

```
lwzu    r4, 4(r3)        # 用更新方式访问 (r3+4)
add     r5, r5, r4       # 保存当前的总和
```

假设r3包含0x10000，它是这个块的开头，第一条语句计算有效地址为0x10004并且将保存在该地址的字（4字节）值载入r4。到目前为止，所有的都是正常的。这个载入完成后，r3的值将被更新为有效地址0x10004。下次邻近4字节存储单元，它是字数组内下一个元素的访问地址，此时r3已经指向了那里。这可以节省很多时间和精力。这使得数组处理或者更一般的相似元素集合的处理变得非常高效。

没有专用指令用于子程序，不存在标准化、硬件增强的系统栈或者系统栈指针概念。取而代之，程序员（更可能，操作系统）借用一个或多个寄存器（通常r1）做栈指针并且使用标准的寄存器/存储器操作来代替压栈和出栈操作。这不仅维护了RISC理念（既然已经完成处理，为什么还要创建专用的出栈指令？），而且允许不同的程序和系统建立不同的栈帧。再者，Apple，IBM和Motorola都有大量的代码基要得到支持，它们对栈的看法不同而且不兼容，所以人们能够明白这个委员会做出这一设计准则的道理。

7.6 性能问题

流水化

如同我们已经研究过的其他芯片，计算机的性能是成功的关键。每个新型计算机都要比以前的快。为了实现这一目标，PowerPC提供了大规模的流水化体系结构（见5.2.4节）。使JVM运行更快的一个不费力方式就是在一个更快的芯片上执行它。为了使PowerPC芯片运行更快些，人们不得不让芯片本身再快些，或者用某种方式在每个系统时钟内压缩更多的计算。为了做到这一点，CPU有一个更加复杂的，流水化的取指-执行周期，允许同时处理几条指令。

RISC体系结构的一个特征就是良好的流水化工作。注意流水线性能的两个关键是保持流水线一直流动以及每个阶段近似均匀。RISC指令集经过特别设计，所有的指令花费同样的时间，并且通常能在单机器周期内执行。所以，在进行取指令的同时机器还能执行加法指令，并且流水线始终保持平稳。这有助于解释PowerPC上有限数目的寻址方式；一条将一个存储单元与另一个相加的指令除了简单的加操作之外还需要4个载入操作和1个保存操作，因此需要4倍的时间（拖延流水线）。PowerPC强迫这个操作被写成4条分开的指令。因为流水化的操作，这些指令仍然在同一时间进行，但可互相协调平稳地完成整个计算，获得良好的性能。

取指令是另一个可以进行优化的方面。对于Pentium，指令长度可变，从1个字节到15个字节。这意味着它取某条指令的时间会是另一些指令的15倍，而当一个非常长的指令被载入时，流水线的其余部分就可能会空闲。相比之下，PowerPC中所有的指令都是等长的并能用单一操作载入，保持流水线是充满的。

当然，有些类型的操作（例如，浮点运算）也需要大量的时间。为了解决这一点，浮点运算的执行阶段本身被流水化（例如，用不同的乘、加以及舍入阶段进行处理），以便它仍然能够以每个时钟周期一条指令的吞吐率进行运算。有时，延迟也是不可避免的，有一种机制可以在必要的时候让处理器停顿，但是一个好的编译器能够编写代码尽可能地最小化延迟。

另一种情况也很容易中断流水线，就是载入不适当的数据，从存储器中不适当的单元取数。在这方面最坏的冒犯者就是条件转移，如“如果小于则转移”。一旦遇到这条指令，下一条指令要么来自于指令序列中邻近的下一条，要么来自于转移目标处的指令——我们可能不知道是哪一个。事实上，我们通常无法判定需要哪条指令，因为条件取决于计算结果，而这

个计算将在流水线中进行，我们目前无法得到计算结果。PowerPC采取设立多个可用的条件寄存器来辅助，有了一个直通的流水线，当转移指令开始执行时，转移目标已经被确定，正确的指令即可被载入。

如同我们将要看到的Pentium，Power体系结构也吸纳了超标量体系结构的成分。实际上，超标量设计与RISC体系结构的关系要比CISC芯片更为紧密，但是好的设计思想就是好的设计思想，往往会被广泛采纳。回顾5.2.5节，这一设计理论吸纳了多条独立不相关指令能在CPU内的不同部件上并行执行的思想。再者，Power体系结构的简化指令集也有助于这一点——算术运算与载入/存储操作，或者比较操作分离，大量寄存器使得一系列指令使用非重叠寄存器（因此能并行化）变得更容易。

典型的PowerPC CPU用独立的模块处理不同种操作（如同早期ALU和FPU之间的区分，只是部件更多）。通常，一个Power芯片至少有一个“整型单元”、一个“浮点单元”和一个“转移单元”（处理转移指令），还可有一个“载入/存储单元”等。PowerPC 603有五个执行模块，分别处理整型运算、浮点运算、转移、载入/存储以及系统寄存器操作。对于更高端的芯片版本，常用单元会在芯片上物理复制；例如，PowerPC G5芯片有10个独立的模块：

- 1个交换单元（完成专门的“交换”操作）
- 1个逻辑运算单元
- 2个浮点运算单元
- 2个定点（寄存器-寄存器）运算单元
- 2个载入/存储单元
- 1个条件/系统寄存器单元
- 1个转移单元

借助这一组硬件，CPU能够同时（在同一取指-执行周期）执行多达10条不同的指令。在指令队列的容量范围内，第一条载入/存储指令将被送至载入/存储单元，而第一条浮点指令将被送至浮点单元，等等。在理论上你能理解下列全部指令完全可以在同一时间内完成：

```
add    r3,r2,r1       # 整数相加
sub    r4,r2,r1       # 另一个定点操作
xor    r5,r5,r5       # 使用逻辑单元清零r5
faddx  r7,r6,r6       # 两个浮点数相加
fsubx  f8,r6,r6       # 使用浮点单元清零r8
b      somewhere      # 转移到somewhere
```

类似地，这些指令中有一半可以在这个周期执行而另外一半在下一个周期进行；或者，这些指令每次执行一条，但是以方便计算机的次序，不一定按照编写的代码次序。

还需要一些特殊的性质才能对程序代码做如此巧妙的处理。首先，指令不可相互依赖；如果上述第2条指令改变了寄存器4的值（它就是这样做的）并且第5条指令需要使用这个新的值，那么第5条指令就至少要到第2条指令结束之后才能开始。然而，借助32个可用的通用寄存器，一个智能编译器通常能找出一种方式在寄存器之间分布计算以便最小化这种依赖关系。相应地，指令应是不同类型的；只有一个逻辑单元，每次就只能执行一条逻辑指令。如果程序的某个部分由连续30条逻辑指令组成，那么该部分被添加到指令队列而且只能每次分派一条指令，基本上将计算机减慢10倍。此外，一个智能编译器会尝试混合代码确保队列中的指令类型不同。

7.7 本章回顾

- PowerPC，由Apple、IBM和Motorola联合设计，是大多数Apple桌面计算机的内置芯片。

它是RISC（精简指令集计算）方法设计CPU的示例，指令数量相对较少，执行速度很快。

- Power体系结构是（主要）来自Motorola和IBM已有设计的灵活折衷。直到近来，PowerPC一直是Apple计算机的基本芯片，Power芯片仍然主导游戏控制台领域。Power芯片实际上是可交叉兼容的芯片系列，其在体系结构的细节中又有很多不同。例如，PowerPC存在32位字长和64位字长两个版本，并且每个芯片与软件交互的方式都极其灵活（例如，任何Power芯片都能以大端和小端格式存储数据）。
- Power CPU有32个通用寄存器和32个浮点寄存器，还有专属芯片的专用寄存器，比Pentium/x86的寄存器要多。该芯片也对几种不同方式的存储管理提供硬件支持，包括对I/O设备的直接（存储器映像的）访问。
- 为了处理速度和简易性，Power所有指令都是等长（32位）的。
- Power汇编语言用3参量格式编写。如同大多数RISC芯片，寻址方式相对较少，数据移入移出寄存器由与算术运算分离的专门载入/存储指令处理。
- PowerPC有若干不同的条件寄存器（CRS），这些条件寄存器能独立地访问。
- 为了加速芯片，PowerPC利用流水化的超标量体系结构执行指令。

7.8 习题

1. RISC芯片与CISC芯片相比优点是什么？缺点是什么？
2. Power体系结构设计的灵活性有哪些实例？
3. 与8088系列相比Power CPU为什么有这么多的寄存器？
4. owerPC算术指令采用3参量格式的优势是什么？
5. and和andi指令之间的区别是什么？
6. and和andi. 操作之间的区别是什么？
7. PowerPC为什么没有标准化的、硬件支持的栈帧格式？
8. 对于流水化来说，为什么所有指令等长很重要？
9. 由JVM提供而不是PowerPC直接提供的指令是什么？PowerPC如何实现这条指令？
10. 重排计算次序是如何加速PowerPC程序的？



第8章 Intel Pentium

8.1 背景

你最后一次是什么时候问计算机价格的？你问的是哪种计算机的价格？对于世界上大多数人，第二个问题的回答可能都是“Pentium”。由Intel制造的Pentium计算机芯片是世界上最畅销的硬件体系结构。即便像AMD这样的竞争对手通常也会非常认真地确保他们的芯片与Pentium的效果一样，连毛病也一样。甚至不运行Windows的大部分计算机（例如，大部分Linux机器）也使用Pentium芯片。

这意味着可预见的未来，如果你必须为一个真实（基于硅片）计算机编写汇编语言程序，可能就会为Pentium编写。为了充分发挥汇编语言的速度，你必须了解这种芯片和它的指令集，以及如何使用。

遗憾地，这是一个复杂的任务，因为“Pentium”本身是一个复杂的芯片，也因为“Pentium”这个术语实际是指一个系列但稍有不同的芯片。最初的Pentium，早在20世纪90年代制造，已经经历了连续的发展，包括Pentium Pro、PentiumⅡ、PentiumⅢ，以及目前的Pentium4 [P4]。期望这个发展继续，毫无疑问Pentium5，6，7将会继续，除非Intel决定改变名称但保持系统兼容（就像80486和Pentium之间所发生的改变）。

这个成功的发展使得学习Pentium体系结构变得既容易又困难。由于最初Pentium（以及先期的x86系列）的巨大成功，已经有了成百万计的程序是为前期的芯片版本编写的，人们仍然想要它们运行在新的计算机上。这就造成了向后兼容的压力，向后兼容要求现代计算机能够毫无问题地执行为原有计算机编写的程序。所以，如果你理解了Pentium体系结构，意味着你理解了大部分P4体系结构（以及x86体系结构）。反过来，每项新措施的提出会增加新的特性而不消除老的特性。这使得Pentium几乎成了CISC体系结构的接收站，因为曾经需要的每个特性依然存在——每个设计决定无论好坏都反映在当前的设计中。不像JVM的设计者可以从零记录开始，Pentium设计者在每个阶段都必须从现有的系统开始，然后对其进行增量地改善。

这使得CPU芯片的基础组织非常复杂，如同一个房子经历了修补，之后再修补，之后再修补。下面就让我们来看一下吧……

8.2 组织和体系结构

8.2.1 中央处理单元

Pentium CPU的逻辑抽象组织与前面描述的8088体系结构非常相像，只是多了些内容。特别是，有更多寄存器，更多总线，更多可选内容，更多指令，总之，每件事都有更多方式去做。同8088一样，仍有8个称为通用的寄存器，但是它们已经被扩展成能容纳32位的量并且有了新的名字。这些寄存器现在是

```
EAX  EBX  ECX  EDX
ESI  EDI  EBP  ESP
```

实际上，EAX寄存器（其他的也如此）只是原来（16位）AX寄存器的一个扩展。如同

AX寄存器被分成AH/AL对，EAX寄存器的低16位就来自8088的AX寄存器。由于这个原因，使用16位AX寄存器的原有8088程序能继续运行在Pentium上。

EAX（32位）			
未使用		AX（16位）	
		AH（8位）	AL（8位）

类似地，16位IP寄存器已经变成EIP寄存器，这个扩展指令指针包含将要执行的下条指令的存储单元地址。不再是原来的4个段寄存器，我们现在有6个段寄存器（CS，SS，DS，ES，FS和GS）用来对经常使用的区域进行存储器访问优化。最后，EFLAGS寄存器保持32个标志，不再是原来的16个标志。除了段寄存器之外，所有这些寄存器都是32位宽。这就意味着Pentium有32位字长，并且大多数操作处理32位的量。

除了这些，8088与Pentium之间一个主要的变化就是建立了不同的操作模式支持多任务。在最初的8088中，任何程序都可自由地访问全部寄存器集，并且通过扩展存储器以及挂接的外部设备——最终，完全控制整个系统及其所有内容。这是有用的。它也可能是危险的，冒着卷入古老的“战争故事”的风险，作者的早期经历是为最初的IBM-PC编写高速图形程序，由于错误地将图形数据放到了硬盘控制器使用的系统存储器区域，当试图使用“被修改”的参数启动时，系统无法正确地工作。

为了防止这类问题，Pentium能在不同的模式下执行，这些模式能控制执行指令的种类，也能控制存储器地址的解释方式。两个特别关注的模式是实模式和保护模式，实模式基本上是8088操作环境的详细翻版（每次运行一个程序，只有1兆字节的存储器可用，没有存储保护），保护模式采用后面（8.4.1节）描述的存储管理系统支持多任务。IBM-PC最初的操作系统MS-DOS运行在实模式下，而MS-Windows和Linux都运行在保护模式下。

8.2.2 存储器

Pentium支持不同的存储器组织和访问方式，但是我们在这里忽略关于它们的大部分内容。首先，它们非常复杂。其次，（从程序员的观点）多数复杂的部分是从古老的8088体系结构继承来的。如果你必须在Pentium上编写模拟8088实模式的程序，才有必要了解这些内容。当你为现代计算机写程序时，任务变得很简单。将存储器视为一个32位地址的扁平组织将会解决用户级的所有必要工作。

8.2.3 设备和外设

Pentium和早期8088的外设接口之间几乎没有重大差别，这也是由于Pentium的兼容性设计。当然，对于计算机厂家而言这是一个巨大的优势，因为用户能够买到新的计算机并且使用原有的外设，在每次升级主板时不必去买新的打印机和硬盘。

然而，随着计算机（和外设）的功能变得愈发强大，新的接口方法开始使用，这就需要设备和I/O控制器去做更多的工作。例如，MS-DOS编程中直接BIOS控制需要标准形式来访问保护模式编程下的非兼容硬件。取而代之，用户级程序对I/O硬件的访问将通过操作系统来控制（并且限定）访问设备的驱动程序请求。

8.3 汇编语言

8.3.1 操作和寻址

Pentium指令集中大部分指令直接从8088继承而来。理论上，从兼容的角度讲，任何为8088编写的程序都可在Pentium上执行。Pentium使用同样的两参量格式，甚至在多数情况下助记符都是相同的。唯一主要的改变就是更新了寄存器助记符来反映扩展（32位）寄存器的使用；指令

```
ADD    EAX, EBX        ; 32位寄存器到32位寄存器的加法
```

是合法的，其功能很明显。

许多新的指令得以创建，明显针对32位的量。例如，针对串原语MOVSB和MOVSW（复制一个字节/字串，前面章节定义的）已经加入了一条新的MOVSD，它将一个双字（32位）串从一个存储单元复制到另一个单元。

8.3.2 高级操作

Pentium也提供了许多全新的操作和助记符，这里不能一一描述。这些指令中很多（也许是大部分）都是执行（常用）任务的捷径，与使用简单的指令相比所需的机器指令数很少。例如，一条指令（BSWAP）交换一个32位寄存器（被指定作为一个参量）的末字节，这个任务使用十多个基本的算术指令才能完成。另一个例子是XCHG（交换）指令，它将源和目的参量互相交换。这些特定操作背后的基本思想是功能强大的编译器能够产生高度优化的机器代码以便最大化系统性能。

这类新指令的另一个例子是新的控制指令，ENTER和LEAVE，它们用来支持高级语言中函数和子程序的语义，这些高级语言包括C、C++、FORTRAN、Ada、Pascal，等。如6.4.7节中所见，这些语言中的局部变量通常放在子程序局部帧系统栈的临时空间。新的ENTER指令在将控制转移到一个新位置时大量地复制了CALL语义，将原有的程序计数值保存在栈中，与此同时，它构建了BP/SP栈帧并且为一组局部变量预留了空间，因此用一条相当复杂的指令替代了半打简单的指令。它也为如Pascal和Ada等语言（但不包括C，C++或FORTRAN）的嵌套声明特性提供了一些支持。LEAVE指令作为单一指令（尽管十分奇特，它不执行从子程序的返回，所以仍然需要RET指令）就能取消栈帧创建。这些指令节省程序代码的字节，但是（与设计者的愿望相悖）作为单一慢速指令它们执行所花的时间要比被它们替代的一组指令执行所花的时间还长。

另外特别增加用来支持高级语言的指令是BOUND指令，该指令检查一个值在两个指定上界和下界之间。要检查的值作为第一个操作数给出，第二个操作数指向两个（邻近）存储单元给出上界和下界。这允许高级语言编译器检查一个数组能够安全地访问。同样，这可用更多传统的比较/转移指令完成，但是要用半打指令并且可能会覆盖一些寄存器。取而代之，CISC指令集用最小的努力让系统完成清晰快速的操作。

多种操作和存储管理模式需要有它们自己的指令组。这样的指令包括VERR和VERW，它们分别验证某个段能够被读出或是写入。另一个例子是INVD指令，它清空内部cache存储器以便确保cache状态与系统主存的状态是一致的。

最后，Pentium不断的发展，从Pentium Pro开始连续经过PⅡ，PⅢ和P4，已经给基本的Pentium体系结构增加了新的能力——也包括新的指令和特性。例如，PentiumⅢ（1999年）增加了一组特殊的128位寄存器，这些寄存器中的每一个都能装入4个（32位）数。增加的新

指令中有（专用的，当然了）同这些寄存器打交道的指令，包括一组SIMD（单指令，多数据）指令，这类指令将同样的操作并行地应用到4个独立的浮点数上。利用这些新的指令和寄存器，浮点性能得以显著地增加（大约4倍），这意味着计算量大的程序如计算机游戏运行速度显著加快——或者说，好的程序员能将4倍高质量的图形包装到游戏中却不会降低其速度。很妙，是不是？

此时，你也许怀疑怎么可能记住所有这些指令。答案是，感谢上帝，并不期待你这样做。Pentium指令集是庞大的，超出了大多数不是每天都与其打交道的人的记忆能力。实际上，只有编译器需要了解这些指令以便能够选择所需要的指定操作。

8.3.3 指令格式

与其他的计算机不同，特别是大多数Power体系结构计算机以及JVM，Pentium不要求所有的指令等长。取而代之，简单的指令——例如，一个寄存器-寄存器的ADD或者从子程序返回指令RET——仅1或2个字节得到保存并解释，可以快速地取出而且在存储器或硬盘中占用的空间很少。复杂的指令可能每条需要高达15个字节。

复杂的指令会包含什么信息？最明显的类型就是立即方式的数据，如（十分明显），32位数据将给任何指令增加4字节的长度。类似地，一个用于间接寻址显示（32位）命名的地址也增加4个字节，所以指令

```
ADD  [EBX], 13572468H      ; 向存储器中加32位的量
```

至少比简单的指令超出4个字节的长度。（在前面的章节中，我们已经明白了典型的RISC芯片如何处理这类问题，基本上将上述指令拆分为多个子阶段。）

除此之外，复杂的指令需要更多的信息。由于多种寻址方式，Pentium要用更多的位（典型的是1个字节，有些是2个字节）来定义哪种位模式将被解释为寄存器、存储器地址等。如果一条指令使用了非标准段（不是缺省的而是为特种指令类型产生的），另外一个可选择的字节就会编码这个信息，该信息包括要使用的段寄存器。用于串操作的多种REP？前缀就用了另外一个字节编码。幸运的是，这些复杂性相对少见，多数指令并不采用。

8.4 存储器组织和使用

存储管理

Pentium存储器最简单的组织是将其作为一个32位扁平的地址空间。每个可能的寄存器模式代表了存储器中一个可能的字节单元。稍大些的模式（由于传统的原因，一个16-位字节模式通常被称为“字”，而一个32位字被称为“双字”）可以增补两个或多个邻近的字节单元。只要计算机能够辨认出多少字节在用，访问长度变化的存储单元就非常简单了。这个方法也非常快，但是有很大的安全隐患，因为任一存储单元，包括操作系统或者在该机器上执行的其他程序使用的存储单元，都可能被篡改。（还记得我的硬盘控制器吗）？所以，这种简单的组织（经常被称为不分段页的存储器）很少使用，除非由于控制器芯片或者其他需要计算机实时、快速地执行任务的应用。

在保护模式下，需要额外的硬件防止程序被非正常修改。从8088继承来的段寄存器（CS，DS等）提供这方面的支持。和8088一样，通用寄存器表示的每个存储器地址将被解释成相对指定段的地址。然而，Pentium中段寄存器内容的解释稍有不同。两个16位用作保护级解释，而其余的14位定义了对这个32位地址的一个扩展，这样创建了一个46位（14+32）有效虚拟或逻辑地址。这就允许计算机访问超出32位（4GB）的地址空间并且将存储区标记为某些程序

不可访问的，以此来保护私用数据。

然而，要通过存储器总线（只有32条，因此用32位地址）来访问，这些46位地址仍然需要被转换成物理地址。从这些虚拟地址到32位物理地址的转换任务由分页硬件处理。Pentium包括一个页表项的目录，该页表项作为这个转换任务的翻译系统。分段和/或分页硬件的使用能被单独地启用或者禁用，这就允许系统用户（或者，更加可能是操作系统的编写者）针对某个应用来调整。从用户的视角，很多复杂的事情都由硬件处理了，所以你只管写你的程序而不需要担心这些。

8.5 性能问题

8.5.1 流水化

改善芯片性能的现代标准技术是流水化。与一些其他的芯片相比，尽管Pentium的指令集不太适于发挥流水化的优势，但是它还是充分地利用了这项技术（如图8-1）。

甚至在Pentium开发之前，Intel 80486就已经采用了5阶段流水线执行指令。5个阶段是：

- 取指：取出指令并将其放入一个预取缓冲区中，预取缓冲区有两个。每个缓冲区都能保存高达16字节（128位），其操作与流水线的其他部分独立无关。
- 译码阶段1：指令译码分为两个阶段进行，第一个阶段作为指令类型的预分析，决定在整个指令中必须提取（以及从哪提取）哪些类型的信息。具体地，D1阶段提取有关操作码和寻址方式的基本信息，不一定涉及地址的细节。
- 译码阶段2：完成指令译码的任务，包括偏移和立即数的辨别，产生ALU控制信号。
- 执行：该阶段执行指令。
- 写回：该阶段用上一阶段产生的结果更新寄存器、状态标志以及cache存储器。

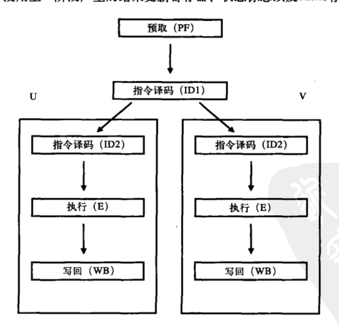


图8-1 Pentium 5阶段流水线示意图，包含超标量U和V流水线

这个流水线比我们已看到的其他流水线（如Power芯片的）更复杂，部分原因是它要处理的指令集的复杂性导致的。注意Pentium（甚至486）指令长度从1到15个字节之间变化。这是预取缓冲区需要这么大空间的部分原因；它们必须能完全容纳下一条指令。也是由于相同的原因，需要花很长时间译码复杂指令——有时和执行一条简单指令时间一样，甚至比这还长。寻址方式的数量和复杂性是主要因素，因为它们为简单的逻辑或算术操作增加了额外的解释层。出于这个原因，就有两个分开的译码阶段以便保持流水线平稳快速地流动。80486能够以接近每个机器周期一个操作的速度执行不含存储器访问的大多数操作。即使如此，间接地址（寄存器中的值作为存储器地址）以及转移都能减慢流水线。

流水化在Pentium及其后来版本中使用相当广泛。不仅有多条流水线（反映超标量体系结构）而且每条流水线还有更多的阶段。初期的Pentium有两条流水线，每条流水线有5个阶段，如上所列。Pentium II 增加了流水线的数量，并且将每条流水线的阶段数增加到12，包括一个判断每条指令长度的特殊阶段。Pentium III 使用14阶段流水线，P4使用24阶段流水线，在此无法做详细讨论。尽管没有为流水线设计有效的指令集，Pentium还是将流水线使用到了极致。

8.5.2 并行操作

为了允许真正的并行操作——在CPU内同时执行两条指令，Pentium采用了其他两个重要的体系结构特性。自Pentium II 开始，指令集以MMX指令为特色专门服务多媒体应用，如高速图形（游戏）或者声音（同样也是游戏）。这些指令实现了SIMD（单指令，多数据）类别的并行，这里对几个独立的数据执行同一个操作。

典型的多媒体应用涉及处理相对少量的数值数据构成的大型数据阵列（例如，屏幕图像中的像素），并对每个数据执行同样的运算，运算速度之快以致于察觉不到图像的闪烁。为了支持这个操作，Pentium II 定义了一组MMX寄存器，每个寄存器都是64位长，保存8个字节，4个字，2个双字，或者不常用的8字节四字。MMX指令定义了在一个寄存器的所有元素上同时执行统一操作。例如，PADDB（字节并行加）指令对8个独立的字节执行8个加法，而PADDW对4个字同时执行4个加法。对于一个简单的操作如显示一幅简单的基于字节的图像，数据处理的速度会是使用普通8位操作的8倍。

8.5.3 超标量体系结构

如前面所讨论的（5.2.5节），另一个重要的并行指令技术涉及流水线阶段的复制，甚至是整个流水线的复制。流水化的难点是保持各阶段的平衡，如果执行一条指令比取指令花的时间长得多，就应将执行的后续阶段重复设置。初期的Pentium使用80486的5段流水线，但是复制了关键阶段，建立了双流水线，称为U流水线和V流水线。指令可以轮流在这两条流水线之间执行，如果两条流水线无障碍，甚至两条同时执行。更一般地，超标量体系结构具有多条流水线或者在流水线的特定阶段有多个执行部件用来处理不同种类的操作（如同早期ALU和FPU之间的区分，这里只是部件更多了）。

Pentium的后期模型，从Pentium II 开始，还在继续扩展。特别指出，Pentium II 的译码阶段1（ID1）或多或少地被复制了三次。理论上，这意味着ID1阶段要花费其他阶段的3倍时长却没有显著减慢整个流水线。实际上，情况非常复杂。第一个指令译码器能处理中等复杂的指令，而第二和第三个译码器只能处理非常简单的指令。因此，Pentium II 硬件包括一个特殊的取指令阶段，它将重新排列指令使其与译码器匹配，所以，指令队列中紧邻的三条指令如果其中之一是复杂指令，就自动将其放在译码阶段1。由于这些复杂的指令非常少见，没有必要考虑给第二和第三个译码器增加这种（昂贵）能力。最后，对于非常复杂的指令，还有第

四个译码器，微码指令序列器（MIS）。

在执行阶段还有一个硬件单元的复制。流水线的一个特殊阶段，重排缓冲器（ROB），接受指令并且将指令分发到多达5组不同的执行单元。这些单元处理多种指令，例如，载入指令，存储指令，整型操作（分为“简单”和“复杂”类型），浮点指令（也有类似划分），以及一些不同的MMX指令。

8.6 再论RISC与CISC

已经花了几章来描述RISC和CISC体系结构之间的区别了，你可能已经注意到实际的区别很小——（RISC）Power芯片和（CISC）Pentium两者采用很多同样的技术来获得机器的最大合理性能。竞争使得两个集团都愿意采用对方好的思想，这是一部分原因。更重要地，随着技术的改进，系列本身变得模糊了。摩尔定律表明放在适度容量微晶片上的晶体管密度以及电路数量正以非常快的速度变得越来越大。反过来，这意味着即使“精简”指令集芯片也能有足够的电路包含常用的指令，即便这些指令是复杂的。

与此同时，硬件已经足够的快，允许非常小规模的软件仿真传统的硬件功能。这个方法，称为微程序设计，意味着在CPU中用其自己的微指令集创建CPU。一条复杂的机器指令——例如，8088/Pentium MOVSB指令将字节串从一个单元移到另一个单元——可以在微指令级实现，通过特有的微指令序列，每次移动一个字节。宏指令将被翻译成数量可能很多的微指令序列，这些微指令来自于程序员不可见的微指令缓冲区，每次执行一条。

这个翻译是Pentium超标量体系结构中ID1译码器的工作。第二和第三个译码器只能将指令翻译成简单的微指令；第一个译码器能够处理更加复杂的指令，产生多达4条微指令。对于更加复杂的指令，MIS作为查找表保存多达上百条微指令，这些微指令针对于Pentium指令集中真正复杂的部分。

在一个更加哲学的层面上，具有讽刺意义的是Pentium（正如所实现的）是一个RISC芯片。各类执行单元核心中的独立微操作正是严格RISC设计核心的小规模、快速的操作。RISC的主要不足——它需要精致的编译器来产生合适的指令集合——不是由精致的编译器/解释器进行处理，而是利用CISC指令集作为近乎中间表达层进行处理。用高级语言编写的程序被编译成CISC指令集可执行文件，然后，在执行时再被重新转换成RISC类微指令。

8.7 本章回顾

- Intel Pentium是一个芯片系列——世界上最著名并且最畅销的CPU芯片。部分归于有效的市场营销，部分归于前期类似芯片如x86系列的成功，Pentium（以及第三方克隆Pentium，如AMD芯片）已经被确立为基于Windows和Linux的计算机选择的CPU芯片。
- 一方面作为CISC设计的产物，另一方面作为对遗留的改进以及向后兼容，Pentium是一个复杂的芯片，拥有庞大的指令集。
- 可用的操作包括通常的算术操作（乘和除有专门的格式，并使用专门的寄存器），数据传送，逻辑操作，以及一些其他专用操作捷径。整个8088指令集都可用，支持向后兼容。
- Pentium包含大量专用指令以便支持特别的操作。例如，ENTER/LEAVE指令支持程序类的操作，源于编译高级语言如Pascal和Ada。
- PentiumⅡ提供了一组多媒体（MMX）指令，用来支持简单算术操作的指令级并行。MMX指令集支持SIMD操作——例如，同时执行8个分开的独立加或者逻辑操作，而不是每次只执行一个操作。

- Pentium也吸纳了广泛应用的流水化和超标量体系结构，提供真正的MIMD并行。
- Pentium的实现包含RISC核，在此，CISC指令是用RISC-类微指令实现的。

8.8 习题

1. 8088和Pentium之间有哪四个主要不同？
2. 指出Pentium中含有但8088中没有的4条指令。
3. 为什么Pentium有两个译码阶段，但是只有一个执行阶段？
4. SIMD并行如何使计算机屏幕上光标/指针平滑移动？
5. Pentium Ⅱ中重排取出的指令的目的是什么？



第9章　微控制器：Atmel AVR

9.1　背景

微控制器是在设备内部用于小规模控制操作的特种计算机，人们通常不认为微控制器是计算机。这类设备的经典案例包括交通灯、烤箱、恒温器以及电梯，但是更好的、更细致的类型是现代汽车中安装的微控制器。例如，防锁死制动由微控制器监视制动系统并且当车轮锁死时（汽车会打滑）切入。微控制器的其他用途包括引爆气囊、调节燃油混合器减少排放，等等。根据Motorola（一家微控制器厂商），仅低端2002型客车就包含了15个微控制器；一个具备更好的娱乐和安全特性的豪华轿车通常有100多个微控制器。这些数字一直以来都在增长。

没有关于微控制器的正式定义，但是它们通常有三个主要特征。第一，它们通常用在所谓的嵌入式系统中，作为一个大系统的部分运行特定的单用途代码，而不是通用可编程的计算机。第二，它们往往是小型、功能较少的计算机。（Zilog Z8 Encore！微控制器使用8位字，运行频率20MHz，只能访问64K的存储器，大约卖$4。与之相比，Pentium4处理器能够轻松地卖到$250以上，而且仅一个裸处理器——没有存储器无法单独使用。）第三，微控制器通常是单芯片器件；它们的存储器和多数外设接口都位于同一个物理芯片上。这在目前很普通，因为几乎所有现代计算机体系结构都有位于CPU芯片上的cache存储器。其重要意义在于一个微控制器可用的存储器根据定义就是全部cache存储器，因此虽然容量小但是速度快。

在本章，我们将详细讨论Atmel公司生产的AVR微控制器。当然，AVR不是这类计算机中的唯一；微控制器是个商品部件，销量数以亿计。这个领域竞争非常激烈；其他制造并销售微控制器的公司包括Microchip、Intel、AMD、Motorola、Zilog、Toshiba、Hitachi和通用仪器。然而，AVR（或者，更精确地，AVR系列，因为有基本AVR设计的若干变形）在其能力方面非常典型，但是与主流芯片如Pentium或者Power芯片（分别由Intel和Apple/IBM/Motorola制造）在令人关注的方面又有所不同。

9.2　组织和体系结构

9.2.1　中央处理单元

考虑到速度和简单性两方面，Atmel AVR采用RISC设计原则。相对很少的指令使得指令长度短（2个字节，相比之PowerPC指令的4个字节，Pentium的指令多达15个字节）并且执行速度快。每条指令都限制在16位标准长度，其中包括必要的参量。指令集针对微控制器通常的需要而定，包括大量用于对单一电气信号操作的位指令。尽管这样，仍然有130条不同的指令（比JVM少）。AVR的指令集甚至也不是最小的；Microchip制造了一个相对普通的极小芯片——用在烤箱上——只有不到35条指令。

Atmel AVR包含32个通用寄存器（编号从R0到R31），还有64个I/O寄存器。这些寄存器每个都是8位宽，对于单一字节或者0～255（或者−128～127）之间的数来说足矣。如同JVM，一些寄存器能够成对使用，以便允许访问大一点的数。不寻常的是（至少与我们已经学过的计算机相比），这些寄存器是物理存储器的一部分，而没有与存储器芯片分离（如图9-1所示）。

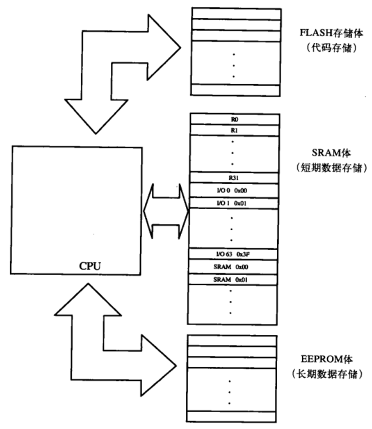


图9-1 AVR体系结构图。特别注意哈佛体系结构（多个分离的存储块）以及不是在CPU中而是在存储器中保存的通用寄存器

考虑所有的实际用途，AVR不支持浮点数；ALU只对整型进行操作，并且是非常小的整数。

从操作上，AVR与我们已经了解的计算机非常相似，有专用指令寄存器、PC、栈指针，等等。

9.2.2 存储器

因为AVR是微控制器，所以AVR上的存储器是非常受限的。与以往不同，AVR的存储器被分成3个独立存储体，这3个存储体不仅在物理特性上不同，而且在大小和容量方面也不同。确切的存储器容量依型号的不同而不同，但是AT90S2313的容量很有代表性。这个机器，如同多数微控制器，是哈佛体系结构设计的范例，这里分开的存储体（和分开的数据总线）可以用于机器指令和数据。借助两个不同路径，每个存储体都能独立地调整发挥其最大性能。更进一步，计算机能同时载入指令和数据（通过两组总线），有效地加倍其速度。但是对于通用计算机，一般也要求两个分开的cache存储器（一个用于指令，一个用于数据），这反过来减少了每个存储器可用的cache容量，于是严重地损害了cache的性能。在微控制器上，存储器就是全部的cache，不存在上述影响。

在AVR中，有三种分开的存储体：一个用于程序代码的只读存储体（因为程序代码在程

序执行过程中不应该被改变)，一个用于程序变量的高速可读/写存储体，第三个存储体用于长期保存程序数据，这些数据必须在断电时还能够保持（例如，日志或者配置信息）。不像传统的体系结构，所有存储器基本上是相同的并且以单一虚拟地址空间来满足任何访问，AVR三个存储体中的每一个都是独立组织并且用它们自己的地址空间和指令来访问。

正如前面讨论的，ROM（只读存储器）和RAM（随机访问存储器）之间存在一个根本的区别，可以从ROM中读，但是不能向ROM中写，而RAM既可读又可写。这个区别在理论上比实际上更重要，随着多种类型存储器的发展，要求大量设备可涂写（一种抽象的、柏拉图的、昂贵的意识）。实际上，ROM/RAM区别的现代定义是使用上的不同：无论CPU能否写人存储体。在AVR中，第一个存储体由FLASH ROM构成。尽管FLASH在理论上是读/写存储器（它是通常用作移动驱动盘的存储器），但是AVR中没有向其写入的电路和指令。从AVR的角度，FLASH存储器提供（在AT90S2313上，2048字节）非挥发存储，断电时也不丢失信息。该存储器，用于所有只读用途，以16位字方式组织，保存可执行的机器代码。特别地，PC中保存的值就是用作指向该存储块的字地址。AVR芯片自身不能对ROM进行任何改变，ROM只能由人们借助合适的外部设备（说实话，设备并不贵）对其重新编程。

相比之下，第二个存储体既可读又可写。这个数据存储器由SRAM（静态随机访问存储器）构成。正如补充资料中讨论的，SRAM和DRAM（动态随机访问存储器）之间的主要区别是动态RAM要求计算机电路给出周期性的“刷新”信号以便保持所存的值。AT90S2313的SRAM容量为256字节。

既然存储器基本上是由最快数据存储电路构成，就不再需要单独的高速寄存器体，如典型计算机中的寄存器。AVR数据存储器组织成一个8位字节序列，分成三个子体。前32字节/字（0x00..0x1F，或者使用Atmel记法为$00..$1F）用来作通用寄存器R0..R31。接下来的64个字（0x20..0x5F）用来作64个I/O寄存器，其余的SRAM提供一组通用的存储器储存单元，用于记录程序变量和/或栈帧。这些存储器储存单元使用从0x60开始直到SRAM的最高地址进行访问（AT90S2313 SRAM的最高地址是0xDF）。

最后，第三个存储体使用另一类存储器，EEPROM。如同FLASH ROM，EEPROM是非挥发的，所以关闭电源后数据也一直保存。与SRAM类似，EEPROM是电可编程的，因此AVR CPU能向EEPROM写数据，并且数据能够永久保存（尽管这需要很长时间，以4ms的数量级）。与SRAM不同的是，数据被写入的次数有限（大约100 000次，尽管技术一直在改进）。这是存储器构造的物理问题，编写AVR程序时应该记住这一点。每秒向EEPROM存储体只写一条数据并且只写一次，一天多一点的时间就会达到100 1000次的写限制。然而，计算机能够从EEPROM安全地读，并且没有读的次数限制。这使得EEPROM是理想的保存现场可修改数据的存储体，这里的数据需要保存但是不用经常修改，例如启动/配置信息或者不频繁的数据日志（比如，每小时一次，这样能记录大约10年）。在我们的例子中，EEPROM存储体容量为128字节。

这些存储体每个都有其自己的地址空间，因此数字0（0x00）不仅能指实际的0也能指FLASH存储器的最低2个字节，EEPROM的最低（单个）字节，或者SRAM的最低字节（即R0）。与所有的汇编语言程序一样，解决这个二义性的关键是上下文；PC中保存的值指向FLASH存储器，而普通寄存器的值（当被读作地址时）指向SRAM中的单元。对EEPROM的访问通过专用硬件，实际上可将EEPROM视为外设。

补充资料

存储器类型

玫瑰到底还是玫瑰，但是存储器不只是存储器。我们已经讨论了几种不同的存储器——例如，RAM和ROM之间的不同。广义讲，工程师们总是讨论不同种类的存储器，每一种都有其适应的用途。

- RAM：随机访问存储器。这是最常用的存储器类型，人们听到存储器时就会想到的；二进制值被当成电信号保存。每组信号集合（通常集合是字或字节尺寸，也可以是单独的位）可以“随机”的次序被独立地访问，因此得名。RAM是一种挥发性存储器，它需要电源来保持信号；如果断电，RAM中所有的信息都会消失。

RAM又分为两种类型：动态RAM（DRAM）和静态RAM（SRAM）。你购买或者看到的大多数存储器是DRAM，它比较便宜并且小巧。每个“位”保存的只不过是一个电容中的电荷（另有一个相关的晶体管），电容在短时内保持能量。DRAM的问题在于存储器电路内部的电荷衰退，即使芯片本身有电源。只要计算机有电SRAM就会记住保存的值而不需刷新。实际上，这意味着计算机必须周期地产生刷新信号来重建DRAM存储模式。（“周期地”在这个上下文中意味每秒几千次。这对于1GHz的处理器来说仍然是一个相当长的时间间隔。）

与之相比，SRAM是自增强的，像附录A中讨论的触发器电路。每个存储位的建立一般需要6～10个晶体管，这意味着一个字节的SRAM存储在芯片上要占用大约10倍的空间，并且成本也高达10倍。另一方面，SRAM不需要刷新周期，所以它可更快地访问。DRAM通常用来做主存储器（这里几百兆字节的额外成本会累加），而SRAM一般用于小型、速度关键的存储应用，如计算机的cache存储器。（出于这个理由，Atmel微控制器只用SRAM做它的可写存储器；还不到1000字节，速度因素超过了价格因素。）尽管SRAM是自刷新的，晶体管也需要电源才能工作，如果电源被切断时间过长（以毫秒计）SRAM的信号就会丢失。因此，SRAM仍被认为是挥发性存储器。

- ROM：只读存储器。RAM既能够读出也能够写入，而ROM芯片不能被写入。更准确的描述是必须要写入ROM芯片时，需要特殊的操作，有时甚至是特殊的设备。然而，ROM的优势在于它是一种非挥发存储器，意味着如果从芯片断电，数据仍能保持。这使得它成为理想的存储器，例如，存储数码相机拍摄的照片。

ROM最简单又古老的组织方式类似于DRAM，但是不使用电容，每个存储单元包含一个二极管，由芯片厂商对其进行编程，仅当存储单元包含1时才允许电流通过。由于二极管不需要电源，而且一般不会消蚀，构建在芯片中的模式永远保持不变，除非你踩踏或诸如此类的破坏。另外，厂商必须确切地知道要用的存储值是什么，如果厂商改变了决定或者出了差错，后果是非常严重的。（1993年Pentium `fdiv`浮点除bug就是ROM出错的一个示例。）

可以非常廉价地大量生产ROM——每个芯片几美分。但是，如果你只需要25个芯片会怎样呢？为此可能就不值得收买整个芯片制造厂。取而代之，使用PROM（可编程的ROM）芯片会更好。像ROM芯片一样，PROM是通过对每个存储单元（不是主动元件，如晶体管）静态电连接而成。在PROM中这些连接实际上是融丝，通过施加足够高的电压可以将融丝熔断。这个过程被形象化地称为“烧制”PROM。PROM一旦被烧制，保持的电气连接（或者是断开）就始终固定不变了。因为不可能再愈合融丝，所以PROM只能被烧制一次。

ROM最后一种形式避免了这个问题。EPROM（可擦写可编程ROM）芯片对于存储元采用先进的量子物理创立基于半导体的可重用融丝。为了建立一个电气连接，存储单元也要像PROM一样接受过电压。然而，应用紫外线光照几分钟之后，“融丝”就会复原。这将擦除整个芯片，允许重新编程以及重新使用。

- 混合存储器 经过获取两方面最佳性能的努力，制造商已经开始生产所谓的混合存储器，混合存储器是电场可编程的，同时还具有ROM的存储优势。EPROM技术的一个变形，如EEPROM（电可擦除可编程ROM）芯片，使用本地电场“擦除”每一个存储单元而不是作为一个整体将芯片全部擦除。由于这个操作是通过电子手段执行的，EEPROM不需要紫外光箱，是个独立部分。

另一方面，写EEPROM要花很长时间，因为存储单元必须暴露在磁场下擦除，这要花费若干毫秒。在理论上，EEPROM与RAM提供同样的功能，因为每个存储单元都可以写、读以及改写，但是计时（以及成本问题）使它并不实用。EEPROM的主要问题在于它们一般只有有限数量的读/写周期。

FLASH存储是加速写入EEPROM过程的一种方法。其基本思想很简单，电场不是单独擦除每个位，而是擦除芯片的大块。擦除一个块与擦除单个位的用时几乎相同，但是在必须大量保存数据时（比如，在一个笔驱动上，像我钟爱的Jump Drive 2.0 Pro，由Lexar Media制造），向存储器传输一个数据块的时间超过了擦写所放置区域的时间。FLASH存储器已被广泛地使用，不仅用于笔驱动，也用做数码相机存储器、智能卡、游戏控制台存储卡以及PCIMCIA卡内的固态盘。

混合存储器的另外一个变形是NVRAM（非挥发RAM），NVRAM实际上是携带电池的SRAM。当电源断开时，电池能够供电使得SRAM芯片中的存储单元保持通电。当然，电池的成本使得这种存储器比简单的SRAM要贵得多。

- SAM（顺序访问存储器）：不提到SAM，对存储器类型的讨论就是不完全的。直觉上任何曾经要在录影带上找出某个情景的人都会熟悉SAM。与RAM不同，SAM只能以特定（顺序）的次序访问，这使得小数据量的使用慢得可怕。多种辅存——CD-ROM、DVD、硬盘，特别是磁带——实际上都是SAM设备，尽管它们一般提供块级的随机访问。

9.2.3 设备和外设

AVR实现了一种简单的存储器-I/O映像。这一设计并不是面向图形量大的环境，在那里人们也许期待以每秒多达20至30次的速度显示由上百万字节组成的画面。取而代之，它只求驱动一个芯片，而这个芯片的输出通过几个引脚物理地（而且电气地）附加到CPU电路。具体地，这些引脚是可寻址的，通过I/O存储体中特别定义的单元对其进行寻址。例如，写人I/O存储单元0x18（SRAM存储单元0x38）被定义成写入“端口B数据寄存器”（PORTB），这等价于端口B第八个输出引脚设置一个电气信号。这个芯片有足够的能源点亮一个简单的发光二极管（LED），或者启动一个电气开关来接通高压电源给更多的用电设备供电。类似地，读取寄存器不同的位能使CPU检测引脚当前的电压，或者检测一个光电池是否对光有反应，或者判定从外部传感器读来的当前温度。

AVR提供若干双向数据端口，这些端口能被单独地定义（基于每个引脚）成输入或输出设备。AVR也提供片上计时电路，可以用来测量经过的时间和/或让芯片对有规律的循环发生

的行为采取措施（如每30秒改变红绿灯或每毫秒读一次发动机的温度）。取决于型号，会有一些嵌入的外设如用于大规模数据发送/接收的UART（通用异步收发器），用来比较两个模拟传感器读数的模拟比较器等。与大型计算机不同，许多输出（引脚）由几个不同的输出设备共享；用于UART的引脚与用于数据端口B的引脚在物理上是相同的。没有这种重叠，芯片在物理上将会非常难用（有上百个引脚需要单独地连接），但是重叠本身意味着几个设备不能同时使用。如果你正在使用端口B，你就不能同时使用UART。

I/O存储器也是CPU本身当前状态信息的保存场所。例如，AVR状态寄存器（SREG）位于I/O单元0x3F（SRAM地址0x5F），并且包含描述CPU当前状态位（如最近计算结果是否为0，或者负数）。栈指针保存在单元0x3D（0x5D）并且定义该单元（在SRAM内）为活动栈单元。由于这些寄存器在编程上被当成存储单元来对待，与I/O外设打交道就如同读写存储单元一样。

9.3 汇编语言

像多数芯片一样，AVR中的寄存器没有任何特别的组织方式。汇编语言指令用两参量格式编写，目的操作数曾是一个源操作数。因此

```
ADD     R0, R1          ; R0 = R0 + R1
```

将R1的值加到R0中，结果在R0中保存，置位SREG反映运算结果。令人迷惑的是，数字0和数字1与更易理解的R0和R1有同样的作用——汇编器对两者都能接受。尽管表面上这非常像一条Pentium或Power的汇编语言指令，但它（当然）符合一个不同的针对Atmel AVR的机器语言标准。

AVR提供了我们希望的大部分普通的算术和逻辑类指令：ADD、SUB、MUL（无符号乘）、MULS（有符号乘）、INC、DEC、AND、OR、COM（按位求反，即NOT）、NEG（补码）、EOR（异或），以及TST（测试寄存器的值，如果该值为0或者是负数则设置相应的位）。也许我们会觉得奇怪，它不提供慢速昂贵的除法操作。取模操作也没有，而且不支持浮点。

为了加速微控制器完成典型计算的速度，有一些-I变形指令（SUBI、ORI、ANOI等等），它们采用立即方式常数并且对寄存器进行操作。例如，

```
ADD     0, 1           ; R0 = R0 + R1
```

寄存器1与寄存器0相加。指令

```
ADDI    0, 1          ;R0 = R0+1, 或者R0增1
```

将立即值1与寄存器0相加。

也有一些指令执行单独的位操作：如，SBR（设置寄存器中的位）或者CBI（清除I/O寄存器中的位）将设置/清除一个通用寄存器或者I/O寄存器中的位。例如，

```
SBR     R0, FF          ; R0 = R0 OR 0xFF
```

将设置寄存器0的低8位为全1。

控制操作也很多。除了转移/跳转指令（无条件：JMP；有条件：BR??，这里??指不同的标志以及SREG中的标志组合；以及跳转到子程序：CALL），还有一些新的操作。SB??操作——第一个?可以是R（通用寄存器）或I（I/O寄存器），第二个?可以是C（清除位）或S（置位），因此SBIC=如果I/O寄存器中的位清除就跳过下条指令——执行一个非常有限的条件转移，如果相应寄存器的位被设置/清除就跳过邻近的下一条指令。AVR也支持间接跳转（无条件：IJMP，对于子程序：ICALL）这里的目标位置取自寄存器（对），具体是保存在R30:R31中的16位值。

9.4 存储器组织和使用

一般的数据传送指令缺省对SRAM存储体进行操作。由于寄存器和I/O寄存器都是这个存储体的一部分，所以向寄存器写入和向一个简单的SRAM字节写入并没有区别。然而，前面一节中定义的算术操作只与通用寄存器（R0..R31）打交道，所以就必须准备着在这个存储体内来回地移动数据了。通常用于此目的的指令是LDS（从SRAM直接载入），该指令第一个参量是寄存器，第二个参量是存储单元，或与LDS相对的SDS（直接存入SDRAM），SDS的参量及处理都与LDS相反。AVR还提供3个特别的间接地址寄存器，X、Y和Z，可用于间接寻址。这些是最后6个寄存器，成对使用（所以X寄存器实际是R26:R27对），用来保存存储器地址。使用这些寄存器和LD（直接载入）指令，代码

```
CLR     R26          ；清除R26（将其设置为0）
LDI     R27, 0x5F    ；给R27载入立即值（常数值）5F
LD      R0, X        ；将（X）[=5F单元中的值]移入R0
```

将存储单元0x005F中的值拷入寄存器0中。它首先将X寄存器的两个部分单独设置为0x00和0x5F，然后将X用作索引寄存器（我们前面已经知道0x5F实际上是SREG寄存器）。做这件事更简单的方式是使用IN指令，该指令从指定的I/O寄存器读取数值：

```
IN      R0, 0x3F     ; 拷贝SREG至R0
```

注意，尽管SREG在存储单元0x5F，它只在I/O端口3F。

使用LPM（从程序存储器载入）指令能间接地访问FLASH存储器（只能读出不能写入）。寄存器Z中的值用作存储器地址，指向程序（Flash）存储区，相应的值被拷入R0寄存器。

访问EEPROM要难得多，主要由于物理和电子的原因。尽管EEPROM存储体在理论上可读/写，但是写入会造成对存储体的物理改变。因此，需要花很长时间完成，并且需要大量的前期准备（尤如发动“电泵”以提供必要的能源进行存储体内容更改），这些准备工作需要在写操作之前进行。AVR设计者选择更接近于类似访问设备的存储器访问模式。

AVR定义了3个I/O寄存器（SRAM中I/O寄存器体的一部分）：EEAR（EEPROM地址寄存器），EEDR（EEPROM数据寄存器）和EECR（EEPROM控制寄存器）。EEAR包含一个与访问地址一致的位模式（在我们的标准示例中，该值在0到127之间，所以最高位应该总是0）。EEDR包含要写入的数据或者是已读出的数据，无论哪种情况都用EEAR中的数据作为目的地址。

EECR包含3个控制位，这些位单独地启用对EEPROM存储体的读或写访问。特别地，EECR的位0（最低有效位）被定义为EERE（EEPROM读启用）位。要从EEPROM给定位置读，程序员应该采取以下步骤：

- 将要访问的字节地址载入EEAR。
- 设置EERE为1，允许读。
- 读出数据。
- 读操作完成后，在EEDR中找出所需数据。

写入的步骤有些复杂，因为有两个启用位需要设置。位2被定义为EEMWE（EEPROM主写启用）位；当该位被置成1时，CPU允许向EEPROM写入。然而，这实际上并没有进行写；这只是做了写的前期准备。真正的写是在EEMWE已经被置为1，然后通过将位1（EEWE/EEPROM写启用位）置为1之后才开始的。经过一段短时（约四条指令的执行时间）之后，EEMWE将自动返回0。这个两阶段提交任务的过程能防止计算机在程序出现bug时意外地改写EEPROM（破坏重要的数据）。

为了向EEPROM写入，程序员应该

• 将要访问的字节地址载入EEAR。
• 将新数据载入EEDR。
• 设置EEMWE为1，允许向EEPROM存储体写入。
• （四个时钟周期之内）设置EEWE为1，允许写入开始。
• 写入数据。

然而，实际写入可能会相当地慢，要花大约4ms的时间。对于一个以10MHz运行的芯片而言，这段时间足以执行40 000（！）个其他操作。由于这个原因，最好不要将EEPROM写放在中间（即一个可能的当前写完成之后等待EEWE位变为0），在EEPROM操作之前，把能做的都做完。

9.5 接口问题

9.5.1 与外部设备的接口

前面描述的EEPROM接口与其他外设接口非常相像。每个设备都由I/O寄存器体内定义的寄存器控制。几个例子就足以表明这种交互。

AT90S2313提供了一个作为嵌入设备的UART来驱动标准串口。我们将忽略电气连接的物理细节，虽然这很有趣，但是它将把我们带到电气工程的领域，而不是计算机体系结构。CPU与UART通过4个寄存器进行硬件交互：UART I/O数据寄存器（保存要被发送或者已经接收的数据）、UART控制寄存器（控制UART的实际操作，例如，启用发送，或者设置操作参数）、UART波特率寄存器（控制数据传输的快/慢），以及UART状态寄存器（显示UART当前状态的只读寄存器）。要通过UART以及串行线路发送数据，首先将要发送的数据载入到UART I/O数据寄存器，然后将所需速率的表示模式载入到UART波特率寄存器。为了执行数据传送，UART控制寄存器一定要被设置成“发送启用”（严格地说，就是UART控制寄存器的位3一定要被置成1）。如果在发送的过程中出现错误，UART状态寄存器的相应位就会被置位，而计算机能够观察到，并且采取适当的校正措施。

与数据端口的交互涉及类似的过程。与UART不同，数据端口可配置成允许多达8个独立的电气信号同时传输。使用该系统，单一数据端口就能同时监视3个按钮（作为输入设备）和一个开关（作为输入设备），还能控制4个输出LED。每个数据端口由两个寄存器控制，一个（数据方向寄存器）用来定义每个位是控制设备输入还是控制设备输出，而另一个（数据寄存器）保持相应的值。要开启与（假如说）引脚6连接的LED，程序员首先应该确认数据方向寄存器的第6位被置1（配置该引脚为输出设备），然后设置数据寄存器的第6位为1，给该引脚高电压（约3~5伏），开启LED。通过设置该引脚电压（接近）至0伏，并且设置这个位为0，将会关掉LED。

补充资料

时钟、时钟速率和定时

计算机怎么知道几点了？更重要地，它们怎样确保需要同时发生的事情（如寄存器中所有的位同时加载）确实能同时发生？一般的回答都会涉及控制时钟或者定时电路。这不过是一个简单电路，通常挂上一个晶体振荡器（像数字钟里的那个振荡器）。这个振荡器每秒振荡几万亿次，每次振荡都被捕获作为一个电信号，并被送往所有的电子器件。技术上，这就是计算机描绘的1.5GHz的来源；这个计算机中主时钟电路产生一个1.5GHz频率

信号，换句话，每秒振荡1 500 000 000次。正如附录A中所见，这个时钟信号允许进行所需计算，防止寄生噪声引入错误。

对于需要反复执行的动作，如每秒刷新屏幕30次，一个受控从电路只要计数（在此情况下）50 000 000主时钟周期，然后刷新屏幕即可。显然，那些需要快速发生的工作，如取指-执行周期，需要尽可能短的计时，理想的是以每时钟周期一次的速率发生。UART控制器的波特率就由类似的受控从电路控制。“速率模式”告诉这个电路在UART必须改变信号之前应该发生多少个主时钟嘀嗒。

这也是超频工作的原理。如果你有一个处理器设计在1.5GHz运行，可以调整主时钟电路（也许甚至更换振荡晶体）运行在2.0GHz。CPU不知道信号到达得太快，它将努力进行高速响应。如果真正以每个时钟周期进行一个取指-执行的速率运行，CPU将尝试更快地取指令，执行。从某种意义上，就像以超出常速放唱片一样（45转/分而不是30转/分）。（可以问问你的父母。）有时这样是可行的，你能廉价地得到30%速度提升。另一方面，CPU也许不能响应这么快的速度，它也许就非常可怕地停止了（例如，如果散热不够充分）。有时CPU会超越其余的部件（要求数据快传而存储器已无法满足它，或者总线无法如此快的传输）。

9.5.2 与定时器的接口

AVR也包括一些内置定时器以便处理通常的任务，如度量时间，或者以规则间隔执行一个操作。（想像一个电梯：开门，电梯等待一个固定的秒数，然后门再关上。只有1毫秒的开门是无用的。）概念上，这些定时器非常简单：一个内部寄存器设置为一个初始值。然后这个定时器统计时钟脉冲（每次给这个内部寄存器加1），这个脉冲要么取自内部系统时钟，要么取自外部定时脉冲源，直到内部寄存器“转了一轮”计数从全1到全0。此时，定时器响起，所计时间已到。

有一个例子放在这里很适合。假设我们有一个时钟脉冲源，每2μs来一个脉冲。给一个8位定时计数器载入一个初值6，每隔2μs计数器将会增至7，8，9，…。经过250次这样的增加（500μs，大约1/2000秒），计数器会转到256，这会溢出使寄存器为0。此时，定时器向CPU发出信号，指示时间已到，以便CPU去做它在等待的事。

一种常用并且重要的定时器就是看门狗定时器，其目标是防止系统死锁。例如，一个写得很差的烤箱程序可能会有bug，它可能在加热功能打开之后就陷入了无限循环。这种无限循环的结果会烧了你的烤箱，很可能还有你的桌子、你的厨房，甚至你的公寓大楼。看门狗是一个普通的定时器工作，只是它发送给CPU的信号等价于按下重启按钮，重新启动系统使之处于一个已知（清醒）状态。要由程序负责周期性地重置看门狗，以防止其触发。

尽管定时器本身很简单，CPU的行动就没有这么简单了。CPU与定时器以两种方式交互。第一，哑方式，CPU把自己放在一个循环中，轮询相应的I/O寄存器，查看定时器是否已完成。如果定时器还没有完成，CPU就返回循环的开始。遗憾的是，这种连续轮询（有时称为忙-等待）的方法防止了CPU完成其他更有用的处理。忙-等待过程可以认为是坐在话机旁等待一个重要电话而不是把握你的生活。

一个更智能的方式处理期待的未来事件（也适于坐等电话）就是建立一个中断处理器。这有些类似AVR如何处理期待却不可预测事件的方式。AVR可以辨识几种中断，这些中断都是在硬件定义环境下产生的，如定时器溢出，规定引脚上的电气信号，甚至是开机。对于

AVR而言，给定芯片的可能中断号从0开始到一个很小的值（如10）。中断向量内这些编号与FLASH ROM（程序代码）中的单元相对应；当中断号0出现时，CPU将跳转到0x00单元并且执行存放在那里的代码。中断号1将跳转到0x01，等等。通常，中断单元内保存的只是一条JMP跳转指令，它将控制转移到一个更大的代码块上，这些代码才是真正执行中断任务的。（特别地，看门狗定时器产生与重启按钮或者加电事件同样的中断，由此可以提供保护，防止无限循环和其他程序bug。）

9.6 设计一个AVR程序

作为最后一个案例，这里给出一个设计（不是全部的代码），展示一个微控制器程序在实际中是如何工作的。第一个结论，微控制器是相当专用的计算机，因此存在很多种程序，以至于为微控制器编写程序有些傻。AVR的物理组织给出了一些明显的警示。例如，在一个没有FPU或者浮点指令的计算机上，编写涉及很多浮点计算的程序是非常愚蠢的。然而，对于其能力范围内的程序而言，AVR是一个非常好的芯片。

作为一个半实际的案例，我们来看一种运行在微控制器上的软件——具体地，关于一个典型的繁忙交叉路口交通灯设计。假设一条北/南方向的街道横穿一条东/西方向的街道，城市交通设计者想要确认每次只有一条街道的交通予以通行。（假设通常红/黄/绿模式，分别意味停止、注意和通行。）

为了让这个系统工作起来，提出4个不同的模式：

模式号	北/南灯	东/西灯	注释
0	绿	红	北/南通行
1	黄	红	北/南缓慢通行
2	红	绿	东/西通行
3	红	黄	东/西缓慢通行

实际上，这也许不能工作。出于安全的考虑，在交通从北/南到东/西切换期间，我们也许想要设置所有交通灯为红，使十字路口空闲，反之亦然。为防紧急情况出现，也需将所有灯置为红灯。我们可以增加额外三种模式：

模式号	北/南灯	东/西灯	注释
4	红	红	北/南即将通行
5	红	红	东/西即将通行
6	红	红	紧急

这个列表产生了另外两个结论。第一，程序必须做的就是模式之间的迁移（通过合适的定时），迁移的次序：0，1，5，2，3，4，0，…。第二，如同许多其他的微控制器程序一样，这个程序没有真正的停止点。只在这种情况下，程序以无限循环的方式运行不仅是有用的，也是必须的。

编写这类程序最简单的方式是实现所谓的状态机。这个程序的"状态"是代表交通灯显示模式的编号（如，如果状态是4，所有的交通灯应是红色）。每个状态能保持一定的时长（由定时器度量）。当定时器中断发生时，计算机将改变状态，然后重新启动定时器以便度量下一段时间。

在这个状态表中，我们也可使用其他的中断。例如，可以加入一个警察专用开关与一个引脚连接，对应一个外部中断。对于这个中断，中断处理器将被写入计算机要进入的具体状态，如所有交通灯变红的紧急状态。当那个开关被合上时（由警察按下按钮），控制器将立即执行这个中断。类似地使用外部中断能导致计算机检查是否有行人按下"步行"按钮，致使计算机迁移到另外一个状态，此时相应的步行灯亮起一段合适的时间。当然，我们可用看门

狗定时器来寻找可能的程序bug，必要时启动它（比如每次灯改变）；万一看门狗定时器触发时，我们可以让程序进入到一个指定的预先编制的普通状态或者是紧急状态，实际上此时有些方面出错，系统需要进行检查。

9.7 本章回顾

- 微控制器是一个小型、单芯片、能力有限的用于小规模操作（如设备控制或者监视）的计算机。
- 微控制器用在很多小器具或设备上，这些看起来根本就不是计算机，如汽车的制动系统。
- Atmel AVR是相关的微控制器系列，具有专用的指令集。AVR体系结构与典型的全方位计算机体系结构有显著不同。例如，AVR不支持浮点操作，包含少于10 000字节的存储器，但是对板级外设有广泛的支持。
- 像很多微控制器一样，Atmel AVR是RISC处理的一个范例。机器指令数量相对很少，具有专用性。
- AVR是哈佛体系结构的一个范例，其存储器被划分为多个（这里是3个）不同的体，每个都有不同的功能和访问方式。AVR的寄存器（和I/O寄存器）位于3个存储体之一，还有专用RAM用来存储变量。AVR有一个FLASH ROM体用于保存程序，以及一个EEPROM体用来存储静态变量，这些静态变量在断电时也要保留。
- Atmel AVR的输入和输出通过存储体内的I/O寄存器完成。一个典型设备有一个控制寄存器和一个数据寄存器。被读出和写入的数据放在数据寄存器中，对控制寄存器中的位进行的操作与要发生的动作相符合。不同的设备有不同的寄存器以及不同的操作。
- 使用中断和相应的中断处理器可以有效地处理不常出现但期待的事件。当一个中断发生时，通常的取指-执行周期改成转移到预先定义的位置，在此处采取针对中断的具体行动。这种中断并不局限于Atmel AVR，而是发生在大多数计算机上，包括Pentium和Power系列。
- 由于硬件和能力的限制，AVR只对于某类程序来说是个好的芯片。一个典型的微控制器程序是永远运行（以无限循环的方式）的状态机，以一个固定的，预先定义的次序执行一组已经定义的行动（类似于改变交通灯）。

9.8 习题

1. 微控制器的三个典型特征是什么？
2. 为什么Atmel AVR在设计中采用RISC原则？
3. 哪种典型的计算机部件在Atmel AVR中却没有？
4. Atmel 是冯诺依曼体系结构的实例吗？为什么？
5. RAM和ROM之间的区别是什么？
6. 为什么SRAM比DRAM每字节的成本要高？
7. Atmel AVR的存储体是什么？它们的用途如何？
8. 看门狗定时器的功能是什么？
9 “存储器-I/O映像”的含义是什么？
10. 描述Atmel所用的让LED反复暗亮的过程。
11. 如果我们想要交通灯在紧急模式下红-暗-红-暗…闪烁，如何修改交通灯示例？

PDG

第10章 JVM高级编程问题

10.1 复杂和派生类型

10.1.1 对派生类型的需求

到现在为止，对计算的讨论都是集中于在基本的类型如整数和浮点数上的操作。但多数问题，尤其是大型或复杂的问题，需要计算机关注非基本类型上的操作。例如，要回答“从巴尔的摩飞行到旧金山的最便宜路线是什么？”这个问题，你就需要了解航班、路线及金额。金额的表示与浮点数密切关联，而路线的表示则与一序列的起始点和停止点密切关联。

从软件设计者的角度看，如果问题的解用高级类型给出的话就容易理解得多，而计算机却只能在其指令集范围内对基本类型进行操作。派生类型的概念很好地填补了这个鸿沟。一个派生类型（derived type）是一个建立在基本类型之上的复杂类型，使得可在其上执行高阶计算。派生类型“金额”从一个浮点数直接建立（或者更精确但不那么直接，可以从各种单位的整数组合建立，这些单位包括美元及美分，或者是英镑、先令、便士以及历史上使用过的法新）。派生类型“地理位置”可由两个分别表示纬度和经度的数字直接建立。

像“路线”这样非常抽象的概念可由“地理位置”之间的“航班”的“链表”建立，每个航班与表示为“金额”的成本相关联。在这个例子中，“路线”就是派生类型。而“链表”、“航班”、“金额”以及“地理位置”也是派生类型，它们最终作为原始类型的适当组合而进行存储和操作。从软件设计者的角度看，如果这样的类型可在计算机系统本身内实现，而且如果程序员能使用这些高级操作的计算机实现，这就是一个很大的优势。派生类型使得程序员和系统设计者能够以比建立单块程序更容易的方式建立复杂的系统。

10.1.2 派生类型的一个例子：数组

理论

最简单和最常用的派生类型之一就是数组（array）。从理论和与平台无关的角度看，数组就是由一个整数来索引的一组相同类型的数据元素。如果必要的话，你可以用这个定义来表示数据结构课程班级的某个人。同时，让我们再进一步解释一下：一个数组就是有时被称为“容器类型”的一个例子，容器类型的意思是其唯一的用途就是存储其他信息块以备后用。在一个数组中，所有的信息块必须是同一数据类型，如必须都是整数或都是字符。当然，它们可以有不同的值。最后，数据的各个位置用一个数，也就是指定了该特定元素的一个整数来寻址。看来还不坏！见图10-1。

```
short figure [ ] = new short [5];
```

如果你有一个1000个整数的数组，那么要占用多大的空间呢？假设是4字节整数，那么理所当然就需要至少4000个字节。不管使用哪种机器都是这样的。然而，这样大的一块存储块不可能装入到一个寄存器中。幸运的是，这不成问题，因为计算要在单个数据元素上而不是在整个数组上执行。这样，程序员就能采用一个策略来得到数据。程序员可相对于数组的基地址（base）来存储数据，该位置也就是数组访问的起始点。在本例中，它对应至少4000字

节长的存储块，通常是数组的起始元素（0号元素）的存储地址。程序员还存储一个偏移（offset），就是他或她想要使用的元素的整数索引。在8088或奔腾中，这些数要存储在两个不同的寄存器中，一个特定的存储模式告诉计算机如何将它们做适当组合来访问数组。

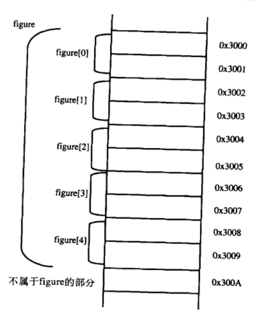


图10-1 显示一个数组的存储布局情况的示意图

在JVM中，情况有点不同，因为"地址模式"不真实存在。实际情况是，两个数被压入到堆栈并执行特定的操作，以此来正确地访问数组。实际上，根据要作什么，有5种特定的操作。大致根据需要程度，这些操作依次是：

- 创建一个新的数组；
- 向数组的一个元素装入一个数据项；
- 从数组的一个元素取得数据；
- 确定数组的长度；
- 当不再需要一个数组时将其销毁。

更重要的是要注意，从高级语言程序员的角度（或从计算机科学理论家的角度）来看，这两种方法之间没有真正的区别。其原因是数组在本质上是由其用法定义的派生类型。只要有执行这些动作的某种方式（例如，向特定索引的数组元素装入一个值），从理论角度看具体实现是无关紧要的。这一点在讨论类时还会谈及。

创建

显然，数组必须在使用之前创建。计算机需要保留适量（可能很大）的存储块。在像C++或Pascal这样的高级语言中，这通常是在数组变量声明时自动完成的。请见下面语句：

```
int sample[1000];
```

该语句声明了sample是一个保存了整数int类型元素的数组变量，并同时保留了能保存从[0]到[999]这1000个整数的足够空间。在Java中，数组的空间是通过显式地数组创建来保留的，比如：

```
int[] sample = new int[1000];
```

这里有点区别。在第一个例子中，数组的创建隐含在声明中。而在第二个例子中，数组的创建（存储块的分配）是通过显式的命令（"new"）来实现的。JVM当然不支持一般意义上的变量声明，但却支持数组创建。这是通过使用机器语言指令newarray来完成的。为了创建一个数组，程序员（及计算机）需要知道要创建的数组大小及其所包含元素的类型。

指令newarray接受单个参数，就是数组元素的基本类型。在这方面汇编器有点奇怪，因为它需要的是类型的Java名字而不是类型表达式。例如，要创建前面定义的数组，类型就是"int"而不是类型表达式I。要创建的数组的长度作为整数必须在堆栈上得到。该长度将被弹出，会为有适当长度和类型的新数组预留空间，然后对应于新数组的地址将压入到堆栈的顶部。这个地址可以适当地装入和处理。

JVM定义前面数组的方法如下：

```
ldc 1000            ; 将1000作为新数组的长度压入
newarray int        ; 创建一个整数数组
astore  1           ; 将新数组存储到#1\vspace*{-4pt}
```

数组是基本类型byte（字节）和char（字符）实际使用的地方。在进行堆栈计算时，字节和字符值自动转换为整数，而在局部变量中它们仍然是32位的量。这有点浪费空间（最多每个局部变量浪费3个字节，一个微小的数量），但这比使JVM存储很小的量浪费空间要少。当数组变大时浪费的空间可能会相当大。事实上，JVM甚至为布尔（Boolean）数组提供一个基本类型，因为机器可选择在一个字节中存储多达8个布尔值（对于字则是32个），这样就将空间效率提高了95%以上。

从技术上说，newarray指令（操作代码0xBC）只创建基本类型如整数和浮点数的一维数组。派生类型数组的创建要更复杂一些，因为还要定义类型。由于派生对象是由程序员有些随意定义的，不存在也不可能存在所有可能派生类型的标准列表。JVM为创建派生类型的一维数组而提供了一条anewarray指令（操作代码0xBD）。如前所述，数组大小必须从堆栈弹出，而类型是作为一个参数给出。然而，参数本身相当复杂（下面要讨论到），实际上是指描述元素类型的常量池中的一个字符串常量。从JVM的角度看，执行操作代码0xBD要比执行0xBC困难得多，因为它要求JVM在常量池中查找这个字符串，解释这个字符串，检验它确实有意义，并有可能加载一个全新的类。从程序员的角度看，创建一个基本类型的数组和创建一个派生类型的数组之间差异很微小，对像jasmin这样的汇编器的一个可能的功能增强就是允许计算机（通过检查参数的类型）确定需要的是操作代码0xBC还是0xBD。

创建派生对象的数组的最困难部分是定义类型。例如，要创建一个字符串类型的数组（见下面），就必须使用完整的类型名"java/lang/String"。下面的例子显示出如何创建1000个字符串的数组：

```
ldc 1000                    ; 数组中有1000个元素
anewarray java/lang/String  ; 创建一个字符串数组
```

这个特定的指令非常不寻常而且是JVM所特有的。大多数计算机不在指令集的层次上提供这类分配存储块的支持，甚至更少的计算机支持有类型块分配的思想。然而，支持有类型计算的能力，甚至当所涉及的类型是由用户定义时，对于JVM设计者所预想的跨平台安全性尤为关键。

除了创建简单的一维数组，JVM还为创建多维数组提供了一条快捷指令。multianewarray指令（注意拼写有点复杂）从堆栈弹出变化的维数，并生成一个适当类型的数组。multianewarray的第一个参数以类型表达式的快捷表示定义了数组的类型（应注意，

不是数组元素的类型），第二个参数定义了维数，也因而定义了需要弹出的堆栈元素的数量。例如，下面的代码就规定要从堆栈弹出3个数，生成一个三维数组：

```
bipush 6              ; 在第三维的数组大小=6
bipush 5              ; 在第二维的数组大小=5
bipush 3              ; 在第一维的数组大小=3
multianewarray [[[F 3 ; 生成一个3×5×6的浮点数组
```

注意，最终生成的数组有三维，总大小为3×5×6。第一个参数“[[[F”将最终数组的类型定义为三维浮点数组（3个[）。

类型系统扩展

从先前对System.out和打印各种内容的讨论中，你应该对系统如何对JVM表达类型有所了解。表10-1的上半部分列出了常见的基本类型（这些类型用在newarray指令及JVM表达式中，并可能在对invokevirtual或multiarray的调用中用到）。除了我们已经熟悉的那些基本计算类型，JVM将字节（B）、短整数（S）、字符（C）以及布尔（Z）也看作是基本类型，这些也列在表10-1中。

派生类型正如所料是由底层类型的表达式派生出来的。例如，一个数组的类型描述就是一个起始方括号（[）后跟数组中每个元素的类型。注意，不需要结束方括号，事实上也不允许。这已经让不止一个程序员感到困惑。整数数组的表达式就是[I，而一个二维浮点数组（矩阵）的表达式就是[[F，这在字面上表达的意思就是：一个数组（[），其中的每个元素都是一个浮点数组（[F）。

表10-1 类型描述表达式

类型	JVM/jasmin表达式
整数（int）	I
长整数（long）	J
浮点数（float）	F
双精度数（double）	D
字节（byte）	B
短整数（short）	S
字符（char）	C
布尔（boolean）	Z
void（返回类型）	V
X类型的数组	[*X*
类Y	L*Y*;
接受X返回Y的函数	(*X*)*Y*

类和类类型用完全限定类名来表达，前面有一个大写的L，后面有一个分号（;）。例如，系统的输出System.out就是一个PrintStream类的对象，存储在java.io（或java/io）目录和包中。这样，System.out的正确的类表达式就是Ljava/io/PrintStream;，这在前面的很多例子中已经用过了。

这些类的构造函数可根据需要结合起来以表示复杂的类型。例如，标准Java对“main”例程的定义要接受一个字符串数组作为参数。在Java中，写作如下的一行代码：

```
public static void main(String [] args)
```

String是系统在java.lang包中定义的一个类，所以会表示成Ljava/lang/String;，而参数就是这种String的一个数组。这个main方法不对调用环境返回任何值，因此声明的返回类型是

void。以下是我们熟悉的迄今在很多方法前面都出现的语句：

```
.method public static main([Ljava/lang/String;)V
```

该语句声明了main方法接受一个String的数组作为唯一的参数而不返回任何值（void）。从这个例子显然可以看出，这个类型系统不仅用于描述数组类型，而且还用于方法的定义，这将在后面讨论。

存储

要存储数组的一项，JVM根据元素的类型提供了若干个类似的指令。为简单起见，暂时假设是整数的数组（[I）。为了在数组中的一个位置存储一个值，JVM需要知道3件事情：哪个数组、哪个位置以及哪个值。程序员必须将这3个值（以这个次序）压入到堆栈上，然后执行指令iastore。这个操作有点不寻常，因为它不向堆栈压入任何东西，所以其纯粹效果就是将堆栈大小减小3。

图10-2给出的简单例子显示出如何将数据存储到数组。这个代码片断首先创建一个10个整数的数组，然后将数0～9存储为相应的数组元素（图10-3）。

```
        bipush 10          ; 需要10个元素
        newarray I         ; 创建一个10个整数的数组
        astore_1           ; 将数组存储到位置#1

        iconst_0           ; 装入一个0，准备循环
        istore_2           ; 并存储到#2

Loop:
        aload_1            ; 装入数组
        iload_2            ; 装入位置
        iload_2            ; 装入值（与位置一样）
        iastore            ; 置array[位置]=值
        iinc 2 1           ; 将1加到#2（位置和值）
        iload_2            ; 我们做完了吗（位置≥10吗）？
        bipush 10          ; 装入10来进行比较
        if_icmplt Loop;    如果还不到10，跳到Loop再重复
        ; 我们做完了！
```

图10-2 在数组中存储的例子

值	地址
0	0x3000
1	0x3004
2	0x3008
3	0x300C
4	0x3010
5	0x3014
6	0x3018
7	0x301C
8	0x3020
9	0x3024

图10-3 图10-2的存储布局

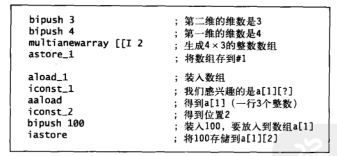
PDG

对于其他的基本类型，包括并非用于计算的字符和短整数，JVM采用在首字母定义变化形式的标准方法来提供适当的类型。例如，要在一个长整型数组中存储一个元素，可使用lastore指令。要在数组中存储一个非基本类型（地址类型，如数组的数组或对象的数组）的元素，元素就存储成地址（a），所以指令就是aastore（见表10-2）。唯一棘手的是布尔类型的数组和字节类型的数组，两个都用b字母打头。幸运的是，JVM本身能处理这个二义性，因为它对字节数组和布尔数组都用bastore指令，并且能自行区分。（的确，这里撒了一个小谎。在大多数的JVM实现中，尤其是来自Sun微系统公司的实现，机器不用费心去区分，因为它就是使用字节来存储布尔数组的元素。这导致每个布尔元素大概浪费7位，但在效率上仍是可接受的）。

表10-2　装入和存储值的数组操作

数组元素类型	存储操作	装入操作
整数（int）	iastore	iaload
长整数（long）	lastore	laload
双精度数（double）	dastore	daload
浮点数（float）	fastore	faload
字符（char）	castore	caload
短整数（short）	sastore	saload
字节（byte）	bastore	baload
布尔（boolean）	bastore	baload
数组（地址）	aastore	aaload
对象（地址）	aastore	aaload

存储一个多维数组必须通过一个存储序列来完成。由于一个多维数组实际上存储成（并视为）数组的数组，首先必须装入相关的子数组，然后在子数组内进行装入和存储。图10-4显示出一个将数100放入到整数矩阵中的一个槽（这里是位置[1][2]）的代码片段：



```
bipush 3                  ; 第二维的维数是3
bipush 4                  ; 第一维的维数是4
multianewarray [[I 2      ; 生成4×3的整数数组
astore_1                  ; 将数组存到#1

aload_1                   ; 装入数组
iconst_1                  ; 我们感兴趣的是a[1][?]
aaload                    ; 得到a[1]（一行3个整数）
iconst_2                  ; 得到位置2
bipush 100                ; 装入100，要放入到数组a[1]
iastore                   ; 将100存储到a[1][2]
```

图10-4　多维数组的创建和存储的例子

装入

装入和存储的原理一样。JVM提供一组?aload系列的指令，用来从数组中取出一个元素。要使用这些指令，就要压入数组和期望的位置。指令将弹出这些参数，然后提取出存储在那个位置上的值并压入，如下所示：

```
aload_1     ; 装入存储在#1的1000个整数的数组
bipush 56 ; 压入值（就是期望的位置）56
iaload      ; 提取出array[56]的整数并压入
```

获取长度

获取一个数组的长度是容易的。arraylength指令弹出堆栈（当然必须是一个数组）的第一项，并压入数组的长度。例如，下面的代码就装入先前创建的样例数组，获得其长度，并将整数值1000留在堆栈上：

```
aload_1          ; 装入先前定义的样例（1000个整数的数组）
arraylength      ; 弹出样例，压入样例的长度
```

销毁

与前面的操作不同，不再需要一个数组的内容时销毁该数组非常简单。JVM标准的定义是，机器本身应该定期地进行垃圾收集（garbage collection），寻找程序不再使用的存储空间、类实例以及变量。一旦发现，就回收并使之可再使用。

对于垃圾收集例程的精确定义和操作，各种JVM实现各有不同。“垃圾”的一般定义是程序不再能达到的存储位置。例如，如果局部变量#1装有一个数组（是数组地址的唯一拷贝），则该数组及其内部的每个元素都是可达到并可用于计算的。如果程序员要对局部变量#1进行写覆盖，则虽然数组本身没有变化，但不可能再访问其内部的信息了。从这一点看，该数组所占用的存储空间（可能很大）不再存储任何有用的东西，就要回收另做他用。唯一的问题就是没有办法精确地预测这种回收可能在什么时候发生，而JVM的实现不完成任何垃圾回收在技术上也是合法的。

从程序员的角度看，没有必要明确地销毁一个数组。弹出或者写覆盖对数组的所有引用（这在程序正常运行过程中经常自动发生），就会导致数组变得不可达到，成为垃圾，并因而被回收。

10.1.3 记录：没有方法的类

理论

下一个最简单的类型称为结构（structure）或者记录（record）。与数组一样，这是一个容器类型。与数组不一样的是，存储在记录中的数据包含在命名的域中，并可以是不同的类型。记录提供了将相关数据一起保存在一个逻辑位置上的方法。如果愿意，你可以将一个记录看作是一个电子棒球交易卡，里面携带了所有与一个球手相关的信息（击球率、本垒打次数、上场击球次数、击球跑垒得分、盗垒数，等等），这些信息以一致的、易于传输的格式存储（见图10-5）。这些信息的每一个部分都与特定的域名关联（例如，“RBI”表示击球跑垒得分），并有不同的类型（击球率定义为浮点数，本垒打次数是整数，而位置“shortstop”（游击手）甚至是一个字符串）。一个更简单的例子是有两个整数域（分子和分母）的分数。

BaseballPlayer类

Name (String)	Babe Ruth
Year (I)	1923
Team (String)	
Games (I)	
At Bats (I)	
Runs (I)	
Hits (I)	
Batting Average (F)	0.393

图10-5 一个部分填写的棒球卡信息记录

与数组不同，每个记录的类型必须在编译时由程序员分别定义。定义主要由一列必要的域名及其各自的类型所组成。这些都存储在一个看上去很熟悉的文件中，如图10-6所示。

这个例子程序显示了记录类型的一个简单实例，就是分数（或者数学家也可能将其看作是有理数）。分数形式上被定义为两个整数的比值，这两个整数分别称为分子（在顶上的数）和分母（在底下的数）。在这个文件中的两个关键行就是用.field汇编指令开头定义了这两个

域。比如，见下面一行：

```
.field public numerator I
```

```
; 'fraction' 类型的结构定义

.class public fraction
.super java/lang/Object

.field public numerator I
.field public denominator I

; 样板——因为“结构”实际上是“类”所以这是需要的
.method public <init>()V
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method
```

图10-6　一个将fraction定义为派生类型的记录样例

该行定义一个名为numerator的域，其值是“I”类型（如前所述，I代表整数）。一个域如果其值是长整数型（long）就用“J”，如果其值是字符串型（String）就用表达式“Ljava/lang/String;”，这些我们前面已经看到。public关键字指出分子值是“public”，意思是它可以被系统中的其他函数和其他对象访问、读以及修改。

文件的其余部分是什么意思？仔细观察你会发现，从类定义的第一个例子开始，我们用的都是同一个样板。其原因很简单：记录在JVM中被实现为一个类，所以要使系统的其余部分能与新定义的记录相互作用，就必须也要有一个最小的类开销。现在没有其他问题了，让我们特别地将类看作是派生类型并了解其优点。

10.2　类和继承

10.2.1　定义类

编程技术（或者更准确地说是程序设计技术）最主要的进步是面向对象（object-orientated）编程技术的发展。在这个框架下，大型程序的设计是通过采用较小的交互式对象（interactive object）系统来实现的，这些对象独立地负责自己的数据处理工作。一个常用的比喻是餐馆。就餐者可点他喜欢的任何菜肴而无须关心做菜的细节，那是厨师的任务。（厨师也无须关心是谁点了某个菜，那是服务员的任务）。这种责任的划分确有好处，因为为某个用途编写的代码片段常常能重新用于其他的用途，而大型系统若看作是一组相互作用的对象来建立就相对容易。作为重复使用的例子，在3.4.1节设计的随机数生成器就可在任何需要随机数的时候使用，不管是做蒙特卡洛模拟、还是维加斯赌博游戏、或者是第1人称枪战游戏。

为了将这些对象结构化，形成一致的框架，通常将它们分组并写成类（class），类就是具有类似性质的对象。两个随机数生成器在很多方面是相同的，即使具体的参数或操作可能不同。特别是，外部的观察者想要对随机数生成器所做的事情（确定种子使之达到一个开始生成随机数的状态，或者从生成器得到一个新的随机数）是相同的。这样我们就可以从操作的角度来定义“随机数生成器”这个概念，也就是说，不是根据它如何工作而是根据我们如何

对其操作来进行定义。任何自称为随机数生成器的东西都要完成这两个操作。

这就得到一个熟悉的类的形式化定义，就是将类（class）定义为派生类型的抽象描述，该派生类型由一组命名的（而不是编号的）并有不同类型的域所组成。（听起来是不是很像是记录？）。不同之处是类还包含方法（method），即定义了与该类交互的合法方式的函数。最后，一个对象（object）是一个类的示例或实例化，所以如果将类看作是一个抽象的概念（如“汽车”），对象就是一个特定的汽车，如我过去在高中常常使用的那辆老式的VM Bug。

快速回顾一下，（从外部来看）类就是将对象聚合起来的一种方式，这些对象具有类似的性质，我们可用相同的方式与其打交道。在一个正在运行苹果的OS X操作系统的计算机上，所有的窗口在左上角都有三个按钮：红色按钮用于删除窗口，黄色按钮用于使窗口变为图标，绿色按钮用于使窗口扩大。一旦用户掌握了如何与任何一个窗口打交道，她就能与所有的窗口打交道。这个思想被推广形成类、对象以及方法的概念。特别是，每个对象（也就是类的实例）共享相同的方法，即所定义的与该对象交互作用的函数。如果你了解了如何驾驭一辆VM Bug，你就能驾驭所有的VM Bug，因为驾驭的“方法”是相同的。

在JVM上编程时，这种观点同样适用，因为两个对象表示成类文件的独立实例，每个类文件在很大程度上是独立的，并依赖于相同的方法在JVM字节码层次上进行通信。我们在前面的程序中已经看到这样的例子，多是在与系统输出进行交互。

详细情况如下：

- System.out（在jasmin中是`System/out`）是一个特殊的对象。
- System/out实例化了`java/io/PrintStream`类。
- 所有的PrintStream对象都有一个`println`方法。
- println方法能使一个字符串出现在“通常”的位置，对于不同对象可以改变这个位置。对于System/out，它出现在诸如屏幕的标准输出中。
- 要做到这一点，可使用`invokevirtual`指令来触发适当对象（System/out）上的适当方法（println）[并且还要以适当的类型（PrintStream）]。

每个系统都要求（通过Java/JVM标准文档）提供一个PrintStream类以及一个System/out对象作为该类的成员。具体的细节（例如，是打印到磁盘文件，还是打印到屏幕窗口，还是打印到打印机）由单独的类确定。而且，建立另一个将其数据打印到不同位置的对象是件容易的事情。这样，要打印到文件而不是到屏幕，你可以不调用System/out的println方法，而是调用新对象的相同方法。

Java不强迫使用面向对象的编程，但为这种语言设计的结构使得使用面向对象编程更容易而且更有利。另外，JVM的结构也不强迫使用面向对象的编程，但确实鼓励使用面向对象编程。特别是，JVM明确地将可执行文件存储成类文件（class file），正如所见，这使得利用交互作用的类建立大规模系统非常容易。我们在下面给出这种派生类的一些例子。

10.2.2 一个简单的类：String

类与数组和域一样是派生类型，但类定义所包括的各种方法使其难理解得多。派生类型的一个典型但相对容易理解的例子是标准的Java String类，定义为Java.lang（或Java/lang）包的组成部分。String类简单地保存一个不可改变的字符串，如“Hello，word！”，这个字符串也许就要被打印。这个类定义了在一个String中要用到的数据类型（通常是字符的数组），也定义了一组方法、函数及操作，它们是与String类进行交互作用的合法途径。我们一直都在使用String对象却没有对其性质有一个正式的了解。

使用一个String

除了存储具体的字符串值，String还支持大量的计算方法，用来检查String以确定其性质。例如，方法charAt()的功能就是接受一个整数并返回其所指定位置的字符。

在Java（或者一个等价的面向对象的高级语言）中，这个函数将被定义成适合调用的方式，就像如下所示：

```
class java.lang.String {
     char charAt(int);
}
```

这个既用于定义也用于调用的方案指出，方法charAt()是类String的一部分（String本身又是java/lang包的一部分），它接受一个整数作为参数并返回一个字符值。在jasmin中，同样的这些概念将用稍微不同的语法来表达，也就是用在表10-1给出的语法来表达。

```
java/lang/String/charAt(I)C
```

（快速复习：这行的意思是符号“charAt”是一个函数，它接受类型I并返回类型C。）我们将会看到，这类语法既用于定义方法本身也用于针对任何特定的串调用方法。

compareTo()方法的功能是比较当前String与另一个String以确定哪一个字母顺序优先。如果this（指当前串）比参数String在字典中靠前，即可能更短或者某字符在通常的排序中更靠前，则返回的就是一个负整数。如果this比String参数更靠后，返回的就是正整数。如果两个String正好相等，则返回0。在jasmin表示法中，这个方法表示如下：

```
java/lang/String/compareTo(Ljava/lang/String;)I
```

属于String类的其他方法还包括：equals()的功能是返回一个布尔值以指示两个串是否相等；EqualsIgnoreCase()的功能是做同样的判断但忽略大小写（所以“Fred”与“fred”不相等但equalsIgnoreCase将为真）；indexOf()的功能是返回作为参数的字符首次出现的位置；length()的功能是返回String的长度；toUpperCase()的功能是返回一个新串，其中的所有字符都已经被转换成大写字母。总共有超过50种不同的方法，这不包括那些隐含地从Object类继承来的方法，这些方法经由标准定义，成为JVM java.lang.String.class的一部分。

10.2.3 实现String

在底层以及在字节码级别，String是如何实现的呢？答案虽然令人烦恼却是有意义的，那就是如何实现无关紧要！任何实现了必要的50种方法的合法JVM类，无论实际的细节如何，都是String类的合法版本。只要其他的类只使用定义明确的标准方法来与String类交互作用，用户就会发现表现很好的String类会满足他们的需要。

例如，实现String的一个（没有价值的）可能性就是看作是对单个字符变量的有序收集，每个字符变量表示String中的单个字符。从程序员的角度看，这有一些明显的缺陷，因为他需要创建很多名字类似seventyeighthcharacter的变量。一个更好的解决方案是采用一个简单的派生类型，如一个字符数组（见前面）。即使在这里也还有一些选择。例如，为了尽量节省空间而采用字节数组（如果设计者预计要处理的String都是ASCII串，这样也就没有必要处理UTF-16串了）。还可以使用整数数组以简化必要的计算。String可在数组中以正常顺序存储，使得数组的第一个元素对应于串的首字符；或者也可用相反的顺序存储，以简化endsWith()方法的实现。

类似地，也可以选择是否创建一个特殊的域，用以将串的长度保存为整数值。如果采用这种做法，就会使每个String对象更大和更复杂一些，但也会使length（）方法使用得更快一些。像这样的权衡取舍或许对于系统的总体性能是重要的，但对于程序员的String版本是否合

法没有影响。其他任何人使用String类时会发现，如果所有方法都有且正确，那么他们的程序总会正确运行。String类文件与使用String的类文件之间没有必要存在特殊的关系。

构造一个String

String有一个特殊的方法，通常称为构造函数（constructor），可用来构造一个新的String。这个方法不接受参数，而且，由于生成的String是零长度且无字符，所以在大部分时间并无太大用处。用迄今本书所使用的符号，这个构造函数描述如下：

```
java/lang/String/<init>()V
```

为了更容易和更有用，String类还有很多（大约11个）其他的构造函数，允许指定要创建的String的内容。例如，程序员可通过复制一个已存在的String来创建新的String：

```
java/lang/String/<init>(Ljava/lang/String;)V
```

也可以从StringBuffer（能从一个可变的字符串构造出不可变的字符串）创建一个字符串：

```
java/lang/String/<init>(Ljava/lang/StringBuffer;)V
```

或者直接从一组字符创建：

```
java/lang/String/<init>([C)V
```

其中的任何一个构造函数都对String的创建及其内容提供控制。

10.3 类的操作和方法

10.3.1 类操作介绍

一般来说，类比数组要困难得多，因为要求类能够（或者至少允许）提供比数组更多的操作和更多种的操作。由于这个原因，随时创建一个新的类通常不现实（虽然总可以通过实例化一个现有的类来创建新的对象）。就像我们一直采用的做法，类是通过.class文件来定义的。类的基本性质，如域和方法，是通过jasmin汇编指令在编译时定义的。这些域和方法一经定义，就可由系统上的任何人使用（要有适当的访问许可）。

10.3.2 域操作

创建域

与数组不同，类中的域带有名字，而不是编号的。然而，不同类的域可能有相同的名字，从而可能引起模糊和混淆。为此，使用域时应该用全称描述，包括该域所在类的名字。

要生成一个域，jasmin使用汇编指令.field，如下所示（在图10-6中也有显示）：

```
.field public AnExampleField I
```

这个例子在当前类中创建一个域，名为AnExampleField，其中包含一个int类型的变量。由于这个类被声明为public（公共）的，所以可被不属于该类的方法访问和操纵，假设程序员有某种理由要这样做。一个更现实的例子（延续前面的棒球示例）可见图10-7。

.field汇编指令允许有若干个其他的访问规范和参数。例如，一个域可声明为final，意思是其值不能由域定义中设定的值所改变；也可以声明为static，意思是该域与一个类而不是类内的单个对象相关联：

```
.field public static final double PI D = 3.1415926537898
```

这个例子将包含π值的“PI”定义为静态的（面向类的）、最终的（不可改变的）双精度数。如果由于某种原因，程序员想要限制对PI的使用只能是该类内部的方法，就可将其声明为private（私有）。有效的访问规范包括：私有（private）、公共（public）、保护（protected）、最终（final）以及静态（static），它们在Java内都有同样的意思。

```
.field public Name Ljava/lang/String;
.field public Year I
.field public Team Ljava/lang/String;
.field public Games I
.field public AtBats I
.field public Runs I
.field public Hits I
.field public BattingAverage F
```

图10-7 假想的BaseballPlayer类的域（见图10-5）

使用域

由于域自动就是其对象/类的组成部分，因此它们作为对象创建的组成部分（或者对于静态域，就是作为类加载的组成部分）被自动创建，而且当相应的对象/类通过垃圾收集被清理时这些域也被销毁。因此对于程序员来说，感兴趣的两个操作就是将数据存储到一个域和从域中取出数据（将其放到堆栈上）。

存储到堆栈的过程类似于存储到数组的过程，主要的不同是域是命名的而不是编号的。这样，putfield操作就接受相关域的名字作为参数（因为这样的名字不能被放置到堆栈上）。程序员需要压入相关对象（其域将要被设置）的地址以及（必须是正确类型的）新值，然后执行适当的putfield指令，如图10-8所示。（执行代码的结果显示于图10-9）。

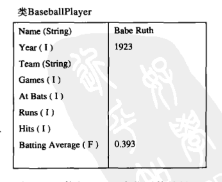
```
aload_1         ; #1应保存BaseballPlayer类的一个实例的地址（即 'a'）
                ; 是类Example的一个对象
                ; 明确地说，是Babe Ruth

ldc "Babe Ruth" ; 压入名字，该名字将被置于Name域中
putfield BaseballPlayer/Name Ljava/lang/String;
aload_1         ; 重新装入对象
                ; 将1923作为整数放入到域中
sipush 1923     ; 压入1923，将要放入到Year域中
putfield BaseballPlayer/Year I    ; 将1923作为整数放入到域中
aload_1         ; 重新装入对象
ldc 0.393       ; 在1923年，Ruth的击球率是0.393
putfield BaseballPlayer/BattingAverage F
                ; 将0.393作为浮点数放入到域中
```

图10-8 在对象域中存储信息

由于静态类属于类而不是特定的对象，所以也有类似的结构，但不需要在堆栈上有一个对象，而是使用putstatic指令简单地置于适当的类域中。如果前面的PI例子没有被定义成“final”，那么程序员只能通过下面语句调节PI的内部值：

```
ldc_2w 3.0          ; 装入3.0，成为PI的新值
putstatic Example/PI D  ; 对Example类的PI置为3
         ; 当然，这不可行，因为PI在前面被定义成 'final' 了
```



类BaseballPlayer

Name (String)	Babe Ruth
Year (I)	1923
Team (String)	
Games (I)	
At Bats (I)	
Runs (I)	
Hits (I)	
Batting Average (F)	0.393

图10-9 执行图10-8中代码的结果

JVM还提供了用于从域中取回值的指令(getfield和getstatic)。System类（在java.lang中定义，因而正式地表达为java/lang/System）包含一个静态定义的名为out的域，这个域包含一个printStream对象。要得到这个对象的值，我们采用

下面已经熟悉的语句：

```
getstatic java/lang/System/out Ljava/io/PrintStream
```

由于java/lang/System是一个类，因此在执行getstatic过程中不需要堆栈上有东西，也不会弹出任何东西。当访问一个非静态域（使用getfield）时，由于值必须从一个特定的对象中选出来，所以该对象必须首先被压入到堆栈：

```
aload_1    ; 从#1装入Example对象
getfield Example/AnExampleField I  ; 将AnExampleField作为整数压入
```

10.3.3 方法

方法介绍

除了域，大多数类还拥有方法，方法就是对存储于类或其对象中的数据的作用方式。方法与域的不同之处在于方法要执行计算，因而包含字节码。与域一样，有若干种具有不同性质的方法。例如，主要的方法必须几乎总要定义成既是public的又是static的，这是由JVM解释器工作的方式所要求的。当JVM试图执行一个类文件时，它要寻找一个方法来执行，该方法是类定义的一部分（不是任何特定对象的组成部分，因为静态类没有对象）。由于该类没有对象，这个方法就必须可公共访问。由于这个原因，我们迄今所写的每个程序都包括下面一行将main定义成public和static方法：

```
.method public static main([Ljava/lang/String;)V
```

通过invokevirtual调用方法

方法在其相应的类文件中声明和定义。要使用一个方法，就必须在适当的对象或类上调用。针对方法调用有一些基本的方法，这些方法根据具体情况以稍微不同的方式使用。

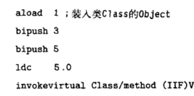
调用方法的最常见和最直截了当的方式就是使用invokevirtual操作（操作代码为0xB6）。我们在多个章节中一直在使用这种方式，没有特别新的内容，概念上也不困难。这个操作从堆栈弹出一个对象（以及方法的参数），调用对象上的适当方法，并压入结果，如下面的标准代码所示：

```
getstatic java/lang/System/out Ljava/io/PrintStream
ldc "Hello, world!"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

这段代码将对象System.out（一个PrintStream）和一个参数压入。通过观察invokevirtual这一行，我们可以看到堆栈顶部必须包含一个java/lang/String类型的参数，并在其下包含一个PrintStream对象。一个更复杂的方法可能要接受若干个参数，但参数仍要在invokevirtual这一行指定，所以计算机能确切地知道要弹出多少个参数。在所有参数之下就是其方法将要被调用的对象（见图10-10）。

当调用这个方法时，控制将以与子例程调用类似的方式传递到新方法。然而，还有一些极为重要的差别。首先，到方法的参数被顺序地置于从#1开始的局部变量中。局部变量#0得到的是其方法将要被调用的对象本身的一个拷贝（用Java的术语，就是#0得到this的一个拷贝）。新的方法还得到全新的一组局部变量和一个全新的（并且是空的）堆栈。

当计算完成后，方法必须有返回，并且以方法定义所期望的适当类型返回。从调用环境的角度看，方法应该向堆栈的顶部压入一个适当类型的元素。从方法的角度看，它需要知道是哪个元素（以及什么类型）。有若干个命令用于返回，首先是不返回任何东西的return，即其


```
aload  1 ；装入类Class的Object
bipush 3
bipush 5
ldc    5.0
invokevirtual Class/method (IIF)V
```

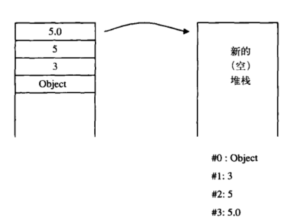


图10-10 使用invokevirtual调用方法

类型是void；?return系列返回堆栈顶部的适当类型的元素。这个系列的成员包括ireturn、lreturn、freturn、dreturn，以及用于返回一个地址或对象的areturn（见图10-11）。然而，这个系列不包括ret指令，该指令是从一个子例程返回。在方法的结尾“离开”也是不合法的。与Java和大多数高级语言不同，在方法的结尾没有隐含的return。

这一点可以从下面一个非常简单的方法中看出来，该方法当且仅当参数是整数3时返回1。

```
.method public isThree(I)I
     .limit locals 2
     .limit stack  2
     iload_1                 ; 参数，是一个整数，在#1中
     bipush 3                ; 装入3，为了比较
     if_icmpeq  Yes          ; 如果#1==3，则返回1
     bipush 0                ; 如果不相等，所以返回0
     ireturn
Yes:
     bipush 1                ; 相等，所以返回1
     ireturn
.end method
```

子例程（通过jsr/ret访问）和方法（通过invokevirtual/?return访问）有一个非常重要的差异。当一个子例程被调用时，调用环境和被调用例程共享局部变量和当前堆栈状态。而每次启动一个方法，你就得到一组全新的（都未初始化的）局部变量和一个全新的（空的）堆栈。由于每次新的方法调用都得到一个新的堆栈和一组新的局部变量，因此，方法以一种子例程没有的方式支持递归（方法调用自身）。要递归地调用方法，可以简单地使用存储在#0中的值作为目标对象，并正常地写invokevirtual这一行。当方法执行时，它将使用其自身的堆栈和局部变量而不影响主计算流程。当从方法返回时，调用环境可简单地从离开的地方重新运行，并使用方法调用的返回结果。

```
bipush  6
ldc     7.0
bipush  1
ireturn
```

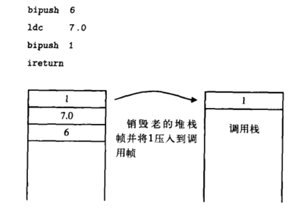


图10-11 采用ireturn的方法返回

其他invoke?指令

由于invokevirtual接受一个对象及其参数来调用一个方法，所以它（或许是显然的）不适于用于静态方法。静态方法没有相关的对象。JVM为调用类上的静态方法提供了一个特殊的invokestatic操作（如图10-12所示）。这个操作与invokevirtual非常类似，只是它不是试图从堆栈弹出一个对象，而是只弹出参数。它不用操心去将对象放入#0，而是用开始于#0的参数填充到局部变量。

```
<init>
bipush  3
bipush  5
ldc     5.0
invokestatic Class/staticmethod(IIF)V
```

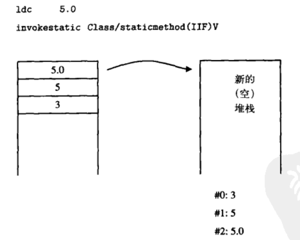


图10-12 使用invokestatic的静态方法调用

还有一些特殊的环境需要特殊处理。特别是当要初始化一个新的对象（使用<init>方法），或者要处理一些涉及超类和私有（private）方法的棘手情况时，就必须要使用一个特殊的操作invokespecial。从程序员的角度看，invokevirtual和invokespecial之间没有差别，但下面标准样板代码说明了invokespecial的主要用途。

```
.method public <init>()V
     aload_0
     invokespecial java/lang/Object/<init>()V
     return
.end method
```

为了初始化一个（任意类的）对象，首先有必要确认它有超类的所有性质（例如，如果Dog（狗）是Animal（动物）的子类，要初始化一个Dog，你就需要确认它是一个合法的Animal)。这是通过调用当前对象（this，或者局部变量#0）上的初始化方法来完成的。但是由于它是一个初始化方法，计算机就不得不使用invokespecial。

声明类

类通常在具有.class扩展名的文件中定义和声明。所以，这些文件就包含了关于类本身及它们与其他类之间关系的必要信息，JVM需要这些信息以保证操作正确。

由于这个原因，每个创建的类文件都需要有两个汇编指令来定义类本身及其在类层次中的位置：

```
.class Something
.super ParentOfSomething
```

像域和方法一样，.class汇编指令也有各种访问规范，如public。类的名字（在上面例子中的Something）应该是完全限定名，包括包的名字。作为java.lang包的一部分定义的类System就包含完全限定类名java/lang/System。类似地，超类应该包括直接超类的完全限定名，而且必须出现。虽然大多数学生写的类都是java/lang/Object的子类，但这不是缺省的，必须明确地给予指定。

还有一些其他的汇编指令在类文件中可有可无，多数是用于源程序调试的汇编指令。例如，.source汇编指令告诉JVM程序用来生成类文件的文件的名字。如果程序在运行过程中出错，JVM解释器能使用这个信息来打印更多有用的错误消息。

10.3.4　类的分类

在上一节讨论中被掩盖的一个重要的微妙之处就是存在若干个不同类型的类文件和方法。虽然它们都非常类似（在存储上的实际差异通常只涉及设置和清除类文件中访问标志的一些位），但它们代表了类语义方面的深刻差异，尤其是从类的外部来看。

在类、对象以及方法之间的最通常的关系中，一个类中的每个对象有其自己的一组数据，但由一组统一的方法来操作。例如，考虑任何特定型号汽车的“类”（让我们特别考虑2007本田雅阁）。显然，这些汽车（在理论上）以相同方式操作，但它们在诸如颜色、油箱油量以及车牌号方面有各自的性质。这些性质在各个Honda对象中被存储成域。另一方面，每辆车的车前灯都用绝对同样的方式控制。打开车前灯的“方法”是类的一个特性，而不是某个汽车的特性。

然而，存在某些作为类总体上的性质，如汽车的长度、轴距宽度以及油箱尺寸。这些性质甚至能直接从设计图中读出，不需要汽车实际存在。我们对两个概念加以区分：类变量(class variable）是指类本身的性质，而实例变量（instance variable）是指各个实例的性质。在JVM中，类变量的域被声明为static并被存储于类本身而不是存储于各个对象：

```
.field color Ljava/lang/String;
.field static carLength D
```

类似地，域可被定义为final以表示其值（无论是一个实例变量还是一个类变量）不能被改变：

```
.field final vehicleIndentification Ljava/lang/String; = "whatever"
.field static final wheels I = 4
.field static final tires I = 4
```

在这个例子中，域vehicleIdentification是一个不变化的实例变量（但不同汽车的值可能不同），而这个类中的所有汽车都有相同固定不变的轮子数（见图10-13）。汽车的长度是该类的一个性质，而汽车的颜色是各个汽车对象的性质并可变化（比如你决定要重新喷漆的话）。

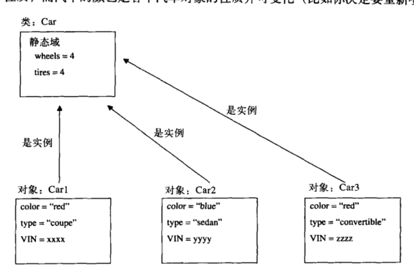


图10-13 类与静态域。所有汽车都有相同数量的轮子，但每个都有其自己的颜色和车牌号（VIN）

实例方法和类方法之间有类似的区别。迄今所写的大多数程序都包括一个如下定义的方法：

```
.method public static main([Ljava/lang/String;)V
```

这个main方法非常重要，因为默认情况是：一旦Java程序运行（或者更准确地说是一个类文件用java执行），Java程序就装入该类文件并寻找一个名字为"main"的方法。如果找到，就试图用一个参数来调用该方法，该参数对应于从命令行键入的其余文字。static关键字指出main方法是与类关联，而不是与任何的类实例关联。因此，就没有必要为了调用main方法而生成适当类的任何实例。

此外，我们还有一个必要的性质"public"，用于指明该方法（或）域可从类的外部访问。一个public方法（或类）对于整个系统都是可见的和可执行的，包括JVM启动序列，而private方法只能从类内的对象/方法来执行。这在"main"本身的结构中当然是隐含的，因为它作为整个系统的第一个方法必须是可执行的。还有其他的访问标志的变型：可以像前面将类、域或方法定义为"final"；或者甚至定义成"abstract"，意思是类本身只是一个抽象，我们不能从中生成实例，而是应该使用该类的一个子类。再则，这些方法和性质的细节与高级Java程序员更为相关，而与理解JVM本身如何工作关系不大。

10.4 对象

10.4.1 作为类的实例创建对象

类定义本身对于完成计算通常用处并不很大。更有用的是作为对象的这些类的实例，也

就是类的具体实例，能够持有数据、调用方法，以及做有用的事情。例如，前面在记录一节定义的“fraction”类简单地定义一个分数有两个域：分子和分母。一个实际的分数就在这些域中有特定的一组值，可用于实际的计算。

要创建一个类的实例，首先必须知道类的名字。请见如下的jasmin语句：

```
new ExampleClassType
```

该语句创建了ExampleClassType类的一个新的实例（在计算机内为其分配存储空间），并将一个地址压入到方法堆栈，该地址指向这个新的对象。仅仅分配空间还不够，还必须要用为该类型定义的某个构造方法将其初始化到一个有用和有意义的状态。要做到这一点，就有必要使用invokespecial并提供适合的方法和一组参数，如图10-14所示。

```
; 示意fraction结构类型使用的程序

.class public fractionfront
.super java/lang/Object

; 样板
.method public <init>()V
      aload_0
      invokespecial java/lang/Object/<init>()V
      return
.end method

.method public static main([Ljava/lang/String;)V

      new fraction
      invokespecial fraction/<init>()V

      return
.end method
```

图10-14　创建一个fraction对象

记住这个分数类（采用标准的样板）定义了一个构造方法<init>，该方法不接受参数。因此，我们构造新的分数就用下面的两行语句来完成分数的创建和初始化。

```
new fraction
invokespecial fraction/<init>()V
```

实际上，这并不十分有用。其原因是，虽然我们刚刚创建和初始化对象，但invokespecial指令在返回时将弹出fraction的唯一引用。结果，我们新分配的存储块会丢失，现在就可能被作为垃圾回收。为了保持新fraction对象的访问，我们需要在调用invokespecial之前复制地址，如图10-15所示。这个图还给出了如何将数据在对象域之间移动的例子。

10.4.2　销毁对象

对象销毁也由垃圾收集系统来处理，并可在指向对象的指针不再处于可访问的位置（如在局部变量中或在堆栈上）时进行。

```
; 示意'fraction'结构类型使用的第二个程序

.class public fractionfront2
.super java/lang/Object

; 样板
.method public <init>()V
      aload_0
      invokespecial java/lang/Object/<init>()V
      return
.end method

.method public static main([Ljava/lang/String;)V
      .limit locals 2
      .limit stack 2

      ; 生成一个新的'fraction'并存储到局部变量1
      new fraction                          ; 生成一个新的'fraction'
      dup                                   ; 复制，以便于call <init>
      invokespecial fraction/<init>()V      ; 初始化
      astore_1                              ; 存储新fraction

      ; 将分子赋值为2
      aload_1                               ; 装入fraction
      iconst_2                              ; 压入分子的值
      putfield fraction/numerator I         ; 将2置于分子（作为整数）
      ; 请注意，不需要对fraction重新存储

      ; 将分母复制为3
      aload_1                               ; 装入fraction（再一次）
      iconst_3                              ; 压入分母
      putfield fraction/denominator I       ; 将3置于分母（作为整数）


      ; 显示分子
      getstatic java/lang/System/out Ljava/io/PrintStream;
      aload_1                               ; 装入fraction
      getfield fraction/numerator I         ; 获得和压入分子
      invokevirtual java/io/PrintStream/print(I)V       ; ……并显示

      ; 显示一个斜线
      getstatic java/lang/System/out Ljava/io/PrintStream;
      ldc "/"
      invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V

      ; 用print(ln)显示分母
      getstatic java/lang/System/out Ljava/io/PrintStream;
      aload_1                               ; 装入fraction
      getfield fraction/denominator I       ; 获得和压入分母
      invokevirtual java/io/PrintStream/println(I)V     ; ……并用print(ln)显示

      return
.end method
```

图10-15 生成和使用fraction对象

10.4.3 类型对象

在任何JVM系统中，基本的类型都命名为“Object”（即对象），或者更全面地说是java/lang/Object。作为整个继承体系的根，系统中的一切都是某种形式的Object。因此，Object的特性就是系统中所有一切所共享的基本性质，Object的方法是可被所有对象调用的方法。这些方法因为是如此地基本，所以并不特别有意思。例如，所有的Object都支持一个equals()方法，该方法在两个对象相同时返回真（“true”），在不同时返回假（“false”）。

由于通用类型的Object可用作一般的数据保存器，程序员就可定义（或者更有可能使用）一个标准化的List类型，用以保存一组Object。然后，无须修改就可用这个类型来保存他的杂货店清单、课表、他最喜爱球队的胜败记录。作为Java语言的一部分，多数的标准数据结构都以这种方式定义，使得其中存储的数据（作为Object）对于如何使用这个结构没有施加任何限制。

10.5 类文件和.class文件结构

10.5.1 类文件

在一个典型的JVM系统中，每个独立的类存储在一个类文件中，类文件就保持了该特定类的使用和执行所必要的信息。多数的信息是显而易见的，例如，类的名称、在继承层次中与其他类的关系，以及类所定义的方法和类的特点。存储的精确细节可能是相当技术化的，甚至可能在不同的JDK版本之间有变化。所以，如果你对类文件格式的细节有专家级的需求（例如，如果你正在写一个编译器，该编译器输出JVM机器指令），你或许就应该去参考一本详细的技术手册，比如像JVM参考规范⊖。对于非专家，下面的描述（以及附录C中更详细的描述）会给出类文件的一些特色。

一般来说，一个类文件存储成一组嵌套的表，如图10-16所示。顶层的表包含了关于类的基本信息，如JVM编译的版本号、类名，以及基本的访问性质。这个表还包含一组子表，包括一个定义了方法、域、属性以及类实现的直接接口子表。另一个子表包含了常量池（constant pool），常量池存储了程序所使用的基本常量值和字符串。例如，如果类每次需要的值是3.1416而不是4字节的浮点值本身，这个值就被置于一个小的常量表中，要用到表的索引。

这就有提高类文件空间效率的效果。虽然程序可用的不同浮点常数超过40亿的，但实际上几乎很少有程序使用的浮点常量数会超过100左右。一个只有200个条目的常量池可用1个字节进行索引，因而每个常量访问就节省3个字节。即使一个使用60000个常量的巨大程序，用2字节索引也能访问任何一个常数。除了存储浮点常量，常量池还保存整数、长整数、双精度数，甚至类似字符串（String）常量的对象（像提示句“Please enter your password（请输入密码）”，可能就存储起来，以便通过调用println打印）。事实上，我们已经熟悉的ldc操作实际上就代表“从常量池中装入”的意思，并（如我们可看到的）可用于几乎任何类型。

⊖ Tim Lindholm和Frank Yellin. The Java Virtual Machine Specification.第2版（Addison-Wesley, Boston, 1999）。

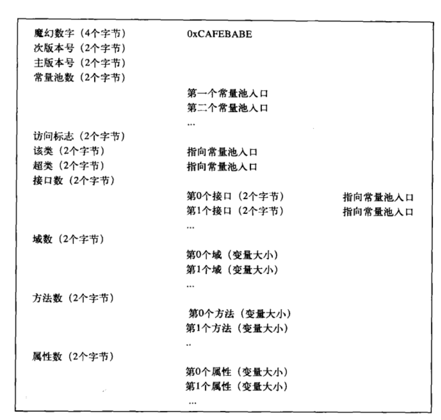


图10-16 类文件格式概览

10.5.2 启动类

在一个类被实际运行程序使用之前，该类必须被从磁盘中加载，链接成可执行的格式，并最终初始化到一个已知的状态。这个过程是JVM程序必须提供的基本服务之一，这种服务是通过原始类加载器（primordial class loader）的形式来提供的，这是标准定义类型java.lang.ClassLoader的一个实例化。JVM设计者在决定原始类加载器在某最小层次上能提供什么服务方面有某些回旋的余地。

例如，要加载一个类，加载器通常必须能在局部存储器中找到这个类（通常通过在类名字的后面加上.class后缀），读入存储的数据，并生成一个java.lang.Class的实例来描述这个类。在某些情况下（如Web浏览器或复杂的JVM实现），可能需要将单个类从归档文件中去除掉，或者从网络上下载适当的applet。原始加载器还必须对类结构足够了解，以便在需要时去除掉超类。如果你写的类扩展了Applet，则JVM要使程序正确运行就需要理解Applet及其特性。将这些不同的类链接（link）成一个适当的运行时的表示则是链接器的任务（通常也结合到类加载器中）。

JVM类加载器的另一个重要任务是验证（verify）字节码执行是安全的。当堆栈上没有整数时，试图执行一个整数乘法就是不安全的。试图对不存在的类创建一个新对象也是不安全的。验证器负责实施本书讨论的大多数安全规则。最后，类通过调用适当的例程来初始化（initialization），将静态域设置为适当值，或者使类成为适合执行的状态。

10.6 类层次汇编指令

这个部分主要是属于Java类层次的高级特性，故可以跳过而不会影响连续性。

实际上，JVM提供的严格类型层次会有一些问题。由于每个子类只能有一个超类，每个对象（或者类）就最多只继承一组性质。真实情况很少如此简洁明确。例如，一个相当明显的层次性就是标准的生物学上的分类。狗是哺乳动物，因而也就是脊椎动物，因而也就是动物，等等。这在JVM类层次中可容易地建立模型，就是使狗成为哺乳动物的子类，依此类推。然而，在实际中狗还继承了宠物类的很多特性，不仅猫、仓鼠、还有古比（一种鱼）、长尾小鹦鹉、鬣蜥（但不包括熊、猎豹以及其他非宠物哺乳动物）等也与狗一样都属于这一类。所以我们可以看到，无论宠物类还包括什么，它都跨越了标准生物学的界限。由于从多个类中继承，也就没有办法在这类结构中创建一个同时是哺乳动物和宠物的类（或对象）。

Java和JVM通过接口（interface）机制提供一个过程，以保持这种规则性。接口就是一个像类一样（事实上，它存储在相同格式的文件中，只有一些访问标志变化）的特殊的对象，它定义了一组性质（域）和函数（方法）。对象永远都不会是一个接口的实例，对象实现接口是通过明确地在其类内部对这些性质和函数进行编码来完成的。例如，可能除了定义宠物的性质，还需要有获得名字和拥有者的函数（假设这些是有字符串返回值的方法），也就是确定名字和拥有者的适当方法。接口永远不会实际地去定义方法的代码，而是定义了程序员所要求实现的方法。类似地，虽然接口能定义域，但作为接口的一部分所定义的域必须是静态的（static）和最终的（final），这反映出一个对象永远都不会是一个接口的实例，这样也就没有因实现域所能得到的存储空间。

到此，一个熟练的程序员就能定义狗（Dog）类，使得其继承哺乳动物（Mammal）类，因而也就得到了哺乳动物的所有性质。他还可以将类定义成"实现"了宠物（Pet）接口，这要确保他为Dog类所写的方法中有每个Pet类都需要的某些方法。

程序员就可以在其类文件中不仅将Mammal类定义成Dog类的超类，还为实现Pet接口做了定义。这就要涉及如下的一个新的汇编指令：

```
.class public Dog
.super Mammal
.implements Pet
```

Pet接口的声明和使用与写一个常规的类文件非常类似，但有两个主要的差异。首先，程序员不是使用`.class`汇编指令来定义类名，而是以相同的方式使用`.interface`汇编指令：

```
.interface public Pet
```

其次，在`Pet.j`文件中的方法会有方法声明但却没有与其关联的实际代码（暗指方法是抽象的）：

```
.method public abstract getName()Ljava/lang/String;
.end method
.method public abstract getOwner()Ljava/lang/String;
.end method
```

接口类型和类类型的用法之间还有一些微小的差异。首先，接口类型不能直接作为对象创建，虽然可以创建实现了接口类型的对象。在上面的例子中，虽然生成一个Dog是合法的，但生成一个Pet却不是合法的：

```
new Dog                ; 可以这样做
new Pet                ; 这会产生一个错误
```

然而，接口可作为参数类型接受并返回方法的值。下面的代码将接受任何实现了Pet接口

的有效对象：

```
invokespecial Dog/isFriendlyTo(LPet;)I
```

该代码还指出，一个特定的Dog是否与一个特定的Iguana（鬣蜥）是友好的。最后，如果你有一个实现了特定接口的对象，但是不知道类是什么（就像上面isFrendlyTo的例子），JVM就提供一个基本的指令invokeinterface，使得接口方法能直接调用而无需考虑底层类是什么。invokeinterface的语法类似于其他的invoke?指令但略复杂一些，通常也比其他方法调用指令更慢和更低效。除非你对接口有特定的需求，最好还是由专家来处理接口问题。

10.7 注释示例：再讨论Hello, World

到此，我们就可以对第一个jasmin例子（包括样板）给出一个详细的解释了：

```
.class public jasminExample
```

是一个类汇编指令，指定了当前类的名字。

```
.super java/lang/Object
```

指出其在标准层次体系中的位置（明确地，是一个java/lang/Object子类）。

```
.method public <init>()V
```

这是构造一个jasminExample类型元素的方法。它不接受参数，无返回值，名为<init>。

```
aload_0
invokespecial java/lang/Object/<init>()V
```

这初始化了一个jasminExample，我们装入例子本身并确保其首先成功地作为一个对象初始化。

```
return
```

无需其他初始化步骤，所以退出该方法。

```
.end method
```

这条汇编指令结束了方法的定义。

```
.method public static main([Ljava/lang/String;)V
```

这是主方法，由Java程序本身调用。它接受一个参数，即一个字符数组，不返回任何值。这个方法定义成公共的和静态的，所以可从类的外部调用而无需存在该类的任何对象。

```
.limit stack 2
```

我们需要两个堆栈元素（一个用于System.out，一个用于String）。

```
getstatic java/lang/System/out Ljava/io/PrintStream;
```

从系统类中获得名为“out”的（静态）域。它的类型应该是PrintStream，在java.io包中定义。

```
ldc "This is a sample program."
```

将要打印的字符串压入（更准确地说，将常量池中字符串的索引压入）。

```
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

在System.out调用“println”方法。

```
return
```

退出这个方法。

```
.end method
```

这条汇编指令结束了方法的定义。

10.8 输入和输出：一个解释

10.8.1 问题描述

除了语言本身的语法，Java编程语言还定义了更多的方面。语言被接受并得到广泛使用的关键是存在各种软件包，用以处理编程中不同方面的困难问题，如输入和输出。例如，java.io就是特别为处理系统输入和输出而设计的一组软件包，该包的处理是在足够抽象的层次上，因而可以在不同的平台上移植。

使用输入和输出外设常常是任何计算机程序的最困难部分之一，尤其是在汇编语言编程中。其原因简单地说就是存在各种令人困惑的设备及其多样的工作方式。

至少在理论上，像磁盘驱动器这样全部输入在任何时候都可得到的设备，与像网络卡这样数据只在特定时间和受限条件下得到的设备有很大的不同。不仅有不同种类的设备，即使是在一个大的种类内部，也有一些微妙但重要的差异，如键盘上按键的布局、是否有第二个鼠标键，等等。然而，从程序员的角度看，这些差异不仅不重要，而且会导致混乱。举个例子，一个电子邮件程序的主要工作是从用户那里读入，得出该邮件要发往哪里，并发送邮件。数据如何到达的细节（比如，是通过一个以太网连接达到吗？是通过键盘按键的一次敲击吗？是通过一个剪切和粘贴操作来得到大块数据吗？是作为硬盘上的一个文件得到吗？）对于电子邮件程序是（也应该是）无关紧要的。

遗憾的是，在基本的机器指令级，这些差异可能是至关重要的。从以太网卡读数据与从磁盘或从键盘读数据不是一回事。（像JVM所支持的）基于类的系统的优势是类本身可被操纵和统一，使得程序员的任务变得不那么混乱。

10.8.2 两个系统比较

一般外设问题

为了说明这种混乱，考虑读入数据并将其打印到某处（到屏幕或磁盘）这样一个任务。为简化，假设输入设备（数据的进入点）是一个配属的键盘或磁盘。不用细解释，键盘就是一个复杂的开关。当一个键被按下时，就产生某个电连接，使得计算机能确定当时哪个键被按下了（因此也就能确定按的是哪个字符）。磁盘是一个信息存储设备。从逻辑上看，你可以将它想象为巨大的一组“扇区（sector）”，每个扇区存有固定数量的字节（通常是512个字节，但也不总是这样）。事实上，还更复杂一点，因为信息被排序成多个一组多盘片（platter）（每个盘片看起来就像一个由磁带制成的CD），每个盘片有若干个编号的同心磁道（track），每个磁道被划分成若干个像比萨切片一样的扇区（sector）（见图10-17）。要读或写一个扇区的数据，计算机需要知道盘片号、磁道号以及磁道内的扇区号。但大多数时间硬盘驱动控制器会处理这些问题。

注意这两种设备之间有一些重要的关键区别。首先，当从键盘读时，由于只能知道在某时刻正在按下的键，你就最多只能得到一个字符的信息。与此相反，从磁盘读却能同时给你一个扇区的信息。在磁盘上还可以向前读以发现下一个扇区装的是什么，而这对于键盘是不可能的。

虽然我们要打印到屏幕的精确细节不同，但都存在类似的差异。如果在计算机上正在运行一个窗口管理器，我们当然就要打印到独立的窗口中而不是像文本模式那样打印到屏幕上。例如，在文本模式中，我们总是知道打印要从实际屏幕的左边开始，但我们完全不知道某个时刻窗口在哪里。我们试图打印时，窗口甚至在移动。因此，根据前面所定义的结构，打印到屏幕与向磁盘写数据全然不是一回事。

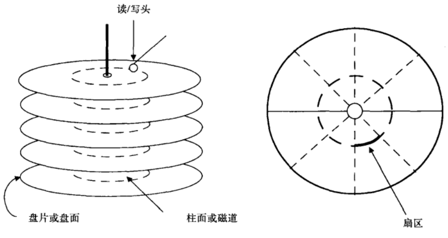


图10-17 磁盘的结构

我们将会看到，JVM通过使用一个适当的类结构而避免了很多混乱。它使用的类结构不同于另外一种常用的系统，就是可使用Windows 98的英特尔奔腾系统。

英特尔奔腾系统

附到奔腾系统的每个设备都带有一套设备驱动器（device driver）。事实上这对于每个计算机都是如此。驱动器定义了如何与硬件交互作用。在这一点上驱动器与类和方法类似，除了没有有用的统一接口。任何Windows系统都有的最简单的一组设备驱动器就是BIOS（基本输入输出系统），作为操作系统的组成部分发售。（实际上，BIOS比奔腾提前近20年就发售，但来自于IBM-PC的顽固影响仍在起作用）。

在英特尔奔腾机上，用一条机器指令（INT）将控制传递给BIOS。当这条指令执行时，计算机检查存储在特定位置（AX寄存器）的值，以确定应该做什么。如果存储的值是0x7305，计算机将访问附加的磁盘。为了正确地做这件事情，它还检查一些存储在其他寄存器中的值。这些值包括：感兴趣的磁盘扇区号、放置新数据的存储位置、从磁盘读还是向磁盘写。在任何一种情况下，都至少要传递一个扇区的数据。

如果存储在AX寄存器中较低半部分的值是0x01，则这条将控制传递到BIOS的指令还导致计算机读一个字符。实际上，这个过程更复杂一些。如果存储的值是0x01，计算机将等待直到一个按键被按下并返回那个字符（还将字符打印到屏幕上）。如果存储的值是0x06，则计算机就检测这时是不是有按键被按下，并返回它（不用打印）。如果没有按键被按下，则不返回任何东西（并将一个特殊标志置位以指明这一点）。这两个功能都是一次只读一个字符。要一次读几个字符，且这些字符在超前敲入缓冲器中可得到的话，就使用存储值0x0A，该功能不返回字符，而是将它们存储到主存储器。

输出有类似的细节问题。要向磁盘写，你就要使用与读入时同样的操作，而在文本模式或窗口系统下写到屏幕需要两种不同的操作（与读操作/值不同）。

所有这些都发生在设备驱动器和硬件控制器已经“简化”了访问外设的任务以后。根本的问题是各种硬件相互之间差异太大。程序员可以写出一个有特殊用途的函数，比如，针对输入完成这样的工作：如果输入要从键盘读入，就使用操作0x01；如果要从磁盘读入，就使用操作0x7305；在任何一种情况下，都将值以统一的格式移动到统一的位置。这样的程序写起来或易或难，但需要对细节的关注、硬件知识，以及不是每个程序员都愿意花费的时间。

这种混乱就是为什么Windows操作系统存在的部分原因。Windows的部分功能就是为程序员提供了粗线条的接口。微软的程序员已经花时间编写了这些考虑了各种情况的详细函数。

JVM

与此形成对照的是，在一个适当设计和构造的基于类的系统中，这种统一是通过类结构本身来达到的。在Java（以及在JVM的扩展）中，大多数的输入和输出都是由类系统来处理的。这就使得类能够处理信息隐藏并只通过方法给出重要的共享性质。

简要回顾一下：java.io.*包是为Java 1.4定义的，它提供了多种类，每个类都有不同的性质用于读和写。用于输入的最有用的、也是最常用的类就是`BufferedReader`。这是一个相当强大的高级别的类，能读入很多种一般化的“流”对象（见图10-18）。Java的1.5版本还包括了一些附加的类，如`Sanner`，也以类似的方式处理。

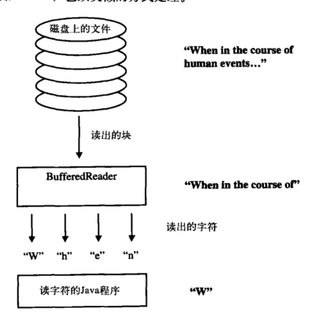


图10-18　使用BufferedReader从磁盘读入一个文件

遗憾的是，键盘缺乏这些BufferedReader对象的很多典型特性，但类系统提供了一种方式从其他种对象来构造一个BufferedReader对象。标准的Java库确实提供了一个对象，这是System类的域，称为`Syatem.in`，通常关联到键盘。然而，这个域被定义成保存最低级、最原始的类型的输入对象之一，即一个java.io.InputStream。（InputStream与BufferedReader之间的关键差异包括缺乏缓冲和除了字节数组不能读其他任何类型）。

java.io.*包还提供了一个特殊的类型用于从磁盘读，无论是作为一个FileInputStream还是作为一个FileReader，两者都可用于构造一个BufferedReader。一旦完成这件事，访问文件就等同于访问键盘，因为两者都使用与BufferedReader类相同的方法。

在下一节中，我们将（采用更广泛可用的Java 1.4语义）构造一个程序，实现的功能是：从键盘读入一行，并将该行拷贝到标准输出。虽然该过程仍复杂，但复杂性在于BufferedReader的构造上而不是在实际的数据读过程。对于对象构造的简单的一次性代价，任何数据都可通过BufferedReader来读，而在英特尔上相对应的程序则对每个输入/输出操作都

需要考虑特殊情况和魔幻数字。

10.8.3 示例：在JVM中从键盘读入

在Java中执行这个任务的代码例子在图10-19中给出。注意需要两个转换：首先是从InputStream构造一个InputStreamReader；其次是从InputStreamReader构造一个BufferedReader。事实上，在Java代码中甚至还隐藏了一点复杂性，因为实际的BufferedReader构造函数是这样定义的：

```
public BufferedReader(Reader in)
```

意思是可从任何种类的Reader来构造BufferedReader，InputStreamReader只是其一个子类。

如前所述，对象的构造是通过两个步骤完成的。首先，必须创建这个新的对象本身（通过new指令），然后必须调用适当的初始化方法（通过invokespecial）。这个程序需要创建两个新的对象，一个是InputStreamReader，一个是BufferedReader。一旦创建了这两个对象，BufferedReader类定义的一个标准方法（称为“readline”）就从键盘读一行并将其作为String返回（Ljava/lang/String;)。采用这种方法，我们就能得到一个串并通过System.out的println方法照常打印。

```
import java.io.*;

class jvmReaderExample {
     public static void main(String[] args) throws IOException {

          InputStreamReader i = new InputStreamReader(System.in);
          BufferedReader b = new BufferedReader(i);

          String s = b.readLine();
          System.out.println(s);
     }
}
```

图10-19 从键盘读入一行并显示的Java样例程序

10.8.4 解答

```
.class public jvmReader
.super java/lang/Object

; 对象创建所需的样板
.method public <init>()V
   aload_0
   invokespecial java/lang/Object/<init>()V
   return
.end method

.method public static main([Ljava/lang/String;)V

.limit stack 4

; 创建一个新的InputStreamReader类型的对象
new java/io/InputStreamReader

; 从System.in (InputStreamReader) 初始化构造函数
dup
```

```
getstatic java/lang/System/in Ljava/io/InputStream;
invokespecial java/io/InputStreamReader/<init>(Ljava/io/InputStream;)V
; 等价于new InputStreamReader(InputStream)
; 现在创建一个新的BufferedReader
 new java/io/BufferedReader
; 复制，并将其置于InputStreamReader之下
dup_x1

; 再次复制，并将其置于InputStreamReader之下
dup_x1
; 堆栈现在保存着BR、BR、ISR、BR

; 消除不需要的BufferedReader
pop

; 使用InputStreamReader调用BufferedReader的构造函数
invokespecial java/io/BufferedReader/<init>(Ljava/io/Reader;)V

    ; 初始化的BufferedStreamReader现在就在堆栈顶部

    ; 调用readline方法得到一个串（并在堆栈顶部离开）
   invokevirtual java/io/BufferedReader/readLine()Ljava/lang/String;

    ; 得到System.out
   getstatic java/lang/System/out Ljava/io/PrintStream;
   swap
   invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

   return
.end method
```

10.9 示例：通过递归求阶乘

10.9.1 问题描述

作为最后一个例子，我们给出在JVM上使用类系统完成递归（即函数或方法调用其自身）的代码。递归函数是在组合学和概率论中广泛使用的数学操作。例如，如果某人想知道洗一副52张的牌有多少种不同的方式，答案就是52！，也就是$52 \times 51 \times \cdots \times 1$。阶乘的一个很好的性质就是有一个简易的递归定义，即：

$$N! = N \cdot (N-1)! \quad 对于N \geqslant 1，且0! = 1$$

采用这个公式，我们就可构造一个递归的方法，就受整数N并返回$N!$。

10.9.2 设计

解决这个问题的方法很简单，可作为一年级编程教科书的一个例子：

```
To calculate N!
     if (N <= 0) then
           return 1
     else begin
           recursively calculate (N-1)!
           multiply by N to get N * (N-1)!
           return N * (N-1)!
     end
end calculation
```

（严格的数学家会指出，这个伪代码还将负数的阶乘定义为1）。

此外，还需要一个（公共的、静态的）主例程来设置N的初值并打印最终结果。由于该主方法是静态的，所以如果阶乘方法不是静态的，就需要创建适当类的一个对象实例。如果我们将阶乘方法定义成静态的，就可直接调用它。

10.9.3 解答

这里给出了计算5！的解。可用类似的代码求解几乎任何递归定义的问题。

```
.class public factorialCalculator
.super java/lang/Object

; 对象创建所需要的样板
.method public <init>()V
   aload_0

   invokespecial java/lang/Object/<init>()V
   return
.end method

.method public static main([Ljava/lang/String;)V
   ; 我们需要两个堆栈元素，一个用于System.out，一个用于String
   .limit stack 2

   bipush 5          ; 压入3，以计算5!
   invokestatic factorialCalculator/fact(I)I
   ; 5!现在就在堆栈上计算出来

   ; 得到和压入System.out
   getstatic java/lang/System/out Ljava/io/PrintStream;

   ; 以准确的次序输出Systen.out和5!
   swap

   ; 调用PrintStream/println方法
   invokevirtual java/io/PrintStream/println(I)V

   return
.end method
.method static fact(I)I
   .limit stack 2

   iload_0       ; 检验是否参数≤0
   ifle  Exit    ; 如果这样，立即返回1

   iload_0       ; 压入N

   iinc 0 -1     ; 计算 (N-1)
   iload_0       ; 装入 (N-1)
   ; 计算 (N-1)!
   invokestatic factorialCalculator/fact(I)I
   ; N和 (N-1) 都在堆栈上
   imul          ; 相乘，得到最终答案
   ireturn       ; 并将答案返回
Exit:
   iconst_1      ; 0!定义为1
   ireturn
.end method
```



10.10 本章回顾

- JVM由于其与面向对象编程语言Java的紧密关联，为面向对象编程和用户定义类的类型提供了直接的支持。对数组和对象的引用均由基本类地址所支持。
- 一个数组是由一个整数或多个整数所索引的一组相同类型数据的集合。有独立的机器级指令来创建、读、写一维和多维数组，还可以获得数组的长度。
- 当数组（或任何数据）不再有用或可访问时，JVM将自动通过垃圾收集来回收所使用的内存，并使其可重用。
- JVM还支持记录（带名字的域的集合）和类（定义了访问方法的记录），并支持接口（抽象方法的集合）。类文件是JVM程序存储的基本单位，它结合了所有这三种类型。
- 域通过`getfield`和`putfield`指令来访问。静态（类）域用`getstatic`和`putstatic`来访问。
- 类通过4种`invoke?`指令之一进行访问，具体使用哪种取决于类和所涉及的方法。
- 对象是作为类的实例由`new`指令创建的。每个类必须包括一个适当的构造函数（典型地命名为<init>），用来将新实例初始化为有效值。
- 在JVM上通过I/O原语访问外部世界是通过一组标准化的类和方法完成的。例如，System.in是一个InputStream类型的静态域，其特性和访问方法由标准文档定义。从键盘读可以用System.in作为一个基类，通过调用特定的方法和创建特定的类来完成。
- 使用类系统也可支持递归，这可通过在每次新的方法调用时生成一组新的局部变量来实现。这不同于以前的`jsr/ret`技术，也不同于像奔腾或PowerPC所采用的在堆栈帧上创建新的局部变量的技术。

10.11 习题

1. JVM的用户定义类型与其他机器如8088或PowerPC在实现方面有哪些不同？
2. 在PowerPC上如何为一个局部数组创建空间？该空间如何回收？在JVM上做法有何不同？
3. 在典型的计算机如奔腾上，数组元素通常是通过索引模式来访问的。在JVM上采用的是什么方法？
4. 像String.toUpperCase()这样的标准方法是如何结合到JVM中的？
5. 域与局部变量有何不同？
6. `invokevirtual`与`invokestatic`有何不同？
7. 为什么静态方法比非静态方法少需要一个参数？
8. 与下列方法对应的类型字符串是什么？
   a. float toFloat(int)
   b. void printString(String)
   c. float average(int, int, int, int)
   d. float average(int [])
   e. double [][] convert(long [][])
   f. boolean isTrue(boolean)
9. 关于<init>方法有什么特殊的地方？
10. 每个类文件（见附录D）中第4个字节是什么？
11. 一个方法能使用的不同字符串值大约有多少？



10.12 编程习题

1. 写一个jasmin程序，采用Java 1.5的Scanner类从键盘读入一行并同时显示在屏幕上。
2. 写一个jasmin程序，采用Java AWT（或一个类似的图形包）在屏幕上显示牙买加国旗。
3. 写一个jasmin程序，确定当前的日期和时间。
4. 写一个Time类以支持时间的算术操作。例如，1:35+2:15为3:50，而1:45+2:25为4:10。
5. 写一个Complex类，支持复数（如3+4i）的加法、减法和乘法。
6. 斐波那契序列可如下递归定义：序列的第一个元素和第二个元素的值都是1。第三个元素是第一个元素和第二个元素的和，即2。一般来说，第N个斐波那契数是第N－1个和第N－2个斐波那契数之和。写一个程序，从键盘读入一个数N，并递归地确定第N个斐波那契数。为什么这是解决这个问题的一个低效的方法。
7. （用教师同意的任何一种语言）写一个程序，从磁盘读入一个类文件，并打印出常数池中的元素数量。
8. （用教师同意的任何一种语言）写一个程序，从磁盘读入一个类文件，并打印在其中所定义的方法的名字。

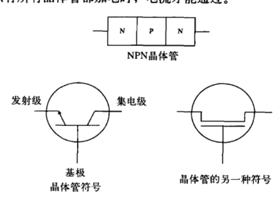


附录A 数 字 逻 辑

A.1 门

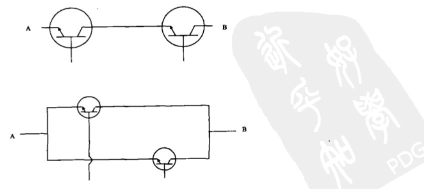
没有晶体管，就没有现代计算机。正如摩尔定律（晶体管密度每18个月增加一倍）所指出的，制造和安置晶体管的能力是在计算机芯片内控制、移动及处理数据的基本方式。

晶体管可被看作是一种电控的开关。一个典型的晶体管及其电子图如图A-1所示。在正常情况下，电流从发射级流到集电极，就像水流过管子或汽车驶过隧道一样。但是，只有在基极施加适当的电信号时，才允许电流流过。没有这个控制信号，就如同关闭了龙头或亮起了红灯。发生这种情况时，电流就不能通过。通过将这些开关结合在一起，工程师们就能建立依赖结构，例如，只有所有晶体管都加电时，电流才能通过。



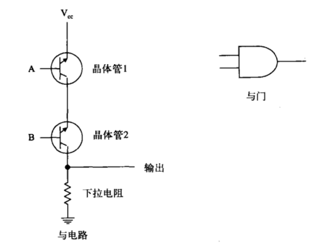
图A-1 晶体管和符号的样例

图A-2显示出两个依赖结构的例子。在串联（serial）电路中，两个晶体管共享一个共同的路径，如果电流要从A点流到B点，就必须能同时流经两个晶体管。在并联（parallel）电路中，每个晶体管有其自身的电流路径，任何晶体管都能允许电流流过电路。

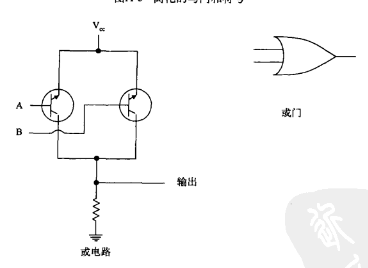


图A-2 串联（上图）和并联（下图）的晶体管

在图A-3中显示的电路是两个晶体管以串联方式互连的例子。为了让电流从电源（V_{cc}）流到输出，对两个晶体管都要施加正确的信号以保证电流通过。只要两个晶体管中任何一个是打开的开关，电流就不能流过。这就意味着只有晶体管1是闭合的并且（AND）晶体管2也是闭合的情况下，电流才能流动（输出端有电力）。类似地，在图A-4中，两个晶体管是并联的，如果第一个晶体管是闭合的或者（OR）第二个晶体管是闭合的（或者两个晶体管都是闭合的），才能允许电流通过。



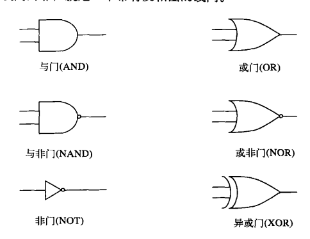
图A-3 简化的与门和符号



图A-4 简化的或门和符号

计算机的基本构件块由称为门（gate）的简单电路组成，这些门实现了这些简单的逻辑信号。每个门一般包含1到6个晶体管，实现了基本的逻辑和算术运算。记住，在逻辑中所用的基本值是真和假。如果“电流在流动”代表真，则图A-3中电路就用一个简单（并且有点理想化，不要用Radio Shack（美国一家著名的消费电子销售商）的零件建造这个系统）的门实现

了逻辑与（AND）的功能。其他能实现的功能还包括：或（OR，见图A-4）、非（NOT）、与非（NAND）以及或非（NOR）。用于绘制各种门的符号在图4-5中给出。注意，每个门（除了非门）都有两个输入和一个输出。还要注意，非门符号在输出线上有一个圈（表示信号反相）。与非门（即与门的非）的符号和与门的符号相同，但在输出处有一个小圈表示反相。类似地，或非门（即或门的非）就是一个带有反相圈的或门。



图A-5 门的类型和符号

A.2 组合电路

复杂的门网络可以实现任何期望的逻辑反应。例如，异或（XOR，即eXclusive OR）的功能是当且仅当只有一个输入为真（而不是两个输入都为真）时，输出才为真。用逻辑表达方式写为：

A XOR B＝（A OR B） AND （NOT（A AND B））

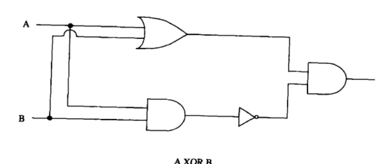
若写成真值表，可如表A-1所示。

表A-1 定义了异或功能的真值表

A	B	结果
真	真	假
真	假	真
假	真	真
假	假	假

最后，要绘制成电路图，则如图A-6所示。

组合电路的基本概念就是输出总是当前输入信号的函数，而无须考虑信号的历史变化。这样的电路对于实现简单的判定或算术操作非常有用。对于如何设计这种电路的全面性描述超出了本附录的范围，图A-7显示出如何用一组门来实现简单的二进制加法（计算机的算术逻辑单元（ALU）的组成部分）。



图A-6 用门实现异或功能

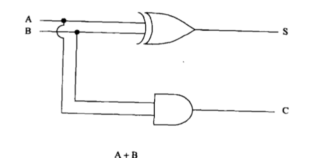
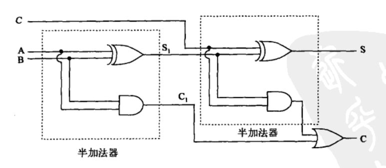
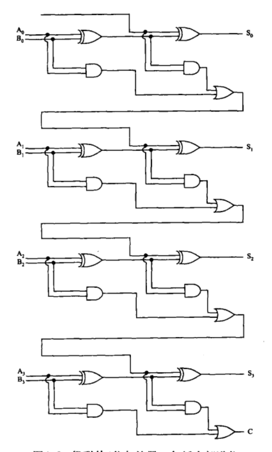


图4-7 单个位的半加法器

明确地说，这个电路要接受两个位信号（A和B），并输出这两个信号的和以及是否有进位。（这个电路有时被称为“半加法器”）。对二进制加法表进行检查可以发现，两个位的和就是对它们做异或运算；而当且仅当两个位都是1时才产生进位，换句话说，就是两个位的与。类似的分析能得出一个更复杂的设计，这个设计能加入前一级加法的进位（就是“全加法器”，见图A-8），或者一次将若干个位相加（就像在寄存器中）。图A-9显示出如何用4个全加法器电路的复制来执行一个4位的加法。当然，32位的复制就能将奔腾的两个寄存器相加。我们也可以建立一个电路来执行二进制乘及其他操作。将这些简单的门增殖几十亿倍，就达到了奔腾的能力和复杂度。



图A-8 单个位的全加法器（带进位）



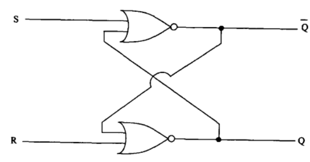
图A-9　级联的4位加法器，包括内部进位

A.3　时序电路

与组合电路形成对照的是，时序电路保留一些存储单元和电路的历史信息。时序电路的输出不仅取决于当前输入，也取决于过去的输入。对此的另一种表达方式就是这种电路有一个内部状态。通过利用这个内部状态，我们就可以存储信息以备后用，实质上就是为寄存器创建必要的存储空间。

一种最简单的时序电路就是S-R触发器（S-R flip-flop），如图A-10所示。为理解这个电路，首先假设S和R都是0（假），Q是0，$\overline{Q}$为1（真）。由于$\overline{Q}$为真，R NOR $\overline{Q}$就是1。类似地，S NOR Q就是1。这样，我们看到这个配置在内部是一致的，只要每个部件都在工作（通常就是指带电），电路就保持稳定。类似的分析还可看出，如果S和R都是0，而Q是1时，结果就是一个自身一致的稳定状态。

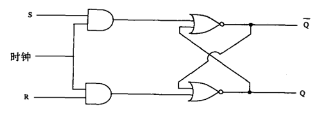
这个简单的触发器可用作为1位存储器，Q的值就是存储在寄存器中的值。信号S和R可分别用于对触发器置位和复位（置为1或0）。我们可以观察到，如果S变为1，将迫使上面的NOR门的输出成为0。在R和$\overline{Q}$都为0的情况下，下面NOR门的输出就为1，所以现在存储的值就是1。类似地分析，如果R置为1，就迫使Q成为1，Q成为0。



图A-10 S-R触发器，带存储器的电路

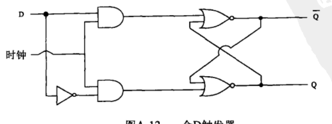
遗憾的是，如果S和R都是1，就会发生不好的事情。根据符号表示，Q和 $\overline{Q}$ 应该总是取相反的值。但如果两个输入都是1，则两个输出就都是0（你可以自行确认一下）。从纯粹的数学意义上说，我们可以将这看作是在逻辑上与用零除等价的情况，即是一种要避免而不是要分析的东西。类似地，即使在输入线上的一个短暂电尖峰也会导致触发器不可预测地改变状态，这是要尽可能避免的事情。

由于要避免这种事情，其他种类的时序电路就更常用于计算机。最常见的电路将控制信号与时序信号（通常称为时钟信号）结合起来，用以同步控制信号并防止短期波动对电路的存储元件产生影响。请注意看图A-11，时钟信号的作用是使触发器能改变状态。如果时钟信号为低，则S或R的变化不能影响电路存储元件的状态。



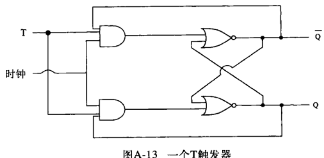
图A-11 一个时钟同步的S-R触发器

我们可以将这个时钟同步的触发器进一步扩展，使其更有用和更安全。图A-12的电路图显示的是一个D触发器。如你所见，这个电路在时钟以外只有一个输入。D输入捆绑到触发器的S输入，而D的互补端 $\overline{D}$ 则被捆绑到R输入。这就避免了S和R同时为1。当一个时钟脉冲发生时，D的值就被存储到触发器（如果D的值是1，则Q变成1；如果D的值为0，则Q变成0）。时钟脉冲发生之前，在D端的变化不会影响触发器的值。这就使D触发器在复制和存储数据方面非常有用。比如，可从I/O设备中读入数据到寄存器以备后用。



图A-12 一个D触发器

SR触发器的另一个变型就是T触发器（图A-13）。像D触发器一样，这是将时钟同步的S-R触发器扩展成单输入，但附加了反馈线，用于控制输入/时钟脉冲如何通过。在这个电路中，每次输入被触发，触发器就改变状态（翻转）。例如，如果Q是1（那么$\overline{Q}$就是0），则下一个脉冲就会触发下面的输入线（本质上就是前面电路的R输入），并复位触发器使得Q为0。



图A-13　一个T触发器

A.4 计算机操作

采用这些构件块就可以建立与计算机体系结构相关的高级结构。特别是，一组1位的触发器（如S-R触发器）就能实现一个简单的寄存器并存储所需数据。例如，要做加法，需要两个1位寄存器，每个寄存器的$\dot{Q}$输出都连接到简单加法电路的一个输入。结果输出位就是这两个寄存器位的和，可捕获并（通过一个D触发器）存储在另一个寄存器中。T触发器可与加法器电路结合使用，以建立一个简单的脉冲计数器。当然，这些描述过于简略，但能够让你对计算机设计者所面临的任务有所认识。



附录B JVM指令集

本文中的示例助记符（操作码用十六进制表示）——说明

概要：本操作码并不存在，只是用来说明每个条目的格式。右侧描述的是相应操作所需的必要初始堆栈状态和创建的最终堆栈状态。注意：long 和double 需要占用两个堆栈单元，如下所示。

初始状态：long

long

最终状态：float

aaload(0x32)——从地址数组中加载值

概要：从堆栈中弹出一个数组地址（对象引用）和integer，取出一维地址数组中指定位置的值。取出的值被压入栈顶。

初始状态：int（索引）

address(数组引用)

最终状态：address（对象）

aastore(0x53)——存储值到地址数组中

概要：将地址（数组引用）存储到同类型的地址数组中。顶端所弹出的参数为定义所用位置的索引。第二个弹出的参数为需存储的地址值，第三个（最后一个）参数为数组本身。

初始状态：int（索引）

address(对象)

address（数组引用）

最终状态：–

aconst_null(0x1)——压入数组常量null

概要：将机器定义的常量null作为地址压入操作数栈中。

初始状态：–

最终状态：address（null）

aload<varnum>(0x19)[byte/short]——从局部变量中加载地址

概要：从第<varnum>个局部变量中加载地址（对象引用）并压入堆栈。在不使用宽(wide)操作数前缀的情况下，<varnum>的值是一个0...255范围的byte。否则为0...65536范围的short。注意：子过程的返回地址不能够通过aload或其他操作码从存储位置中加载。

初始状态：–

最终状态：address

aload_0(0x2a)——从局部变量0加载地址

概要：从局部变量0中加载地址（对象引用）并压入堆栈中。此功能等价于 aload 0，但它占用的字节数更少且速度更快。

初始状态：–

最终状态：address

aload_1(0x2b)——从局部变量1加载地址

概要：从局部变量1中加载地址（对象引用）并压入堆栈中。此功能等价于aload 1，但它占用的字节数更少且速度更快。

初始状态：-

最终状态：address

aload_2(0x2c)——从局部变量2加载地址

概要：从局部变量2中加载地址（对象引用）并压入堆栈中。此功能等价于aload 2，但它占用的字节数更少且速度更快。

初始状态：-

最终状态：address

aload_3(0x2d)——从局部变量3加载地址

概要：从局部变量3中加载地址（对象引用）并压入堆栈中。此功能等价于aload 3，但它占用的字节数更少且速度更快。

初始状态：-

最终状态：address

anewarray<type>(0xbd)[int]——创建一维对象数组

概要：为类型为<type>的一维数组分配空间并将新数组的引用压入堆栈。字节码中的type为常数池中的4字节索引。新数组的大小由从栈顶弹出的一个integer来表示。

初始状态：int（大小）

最终状态：address（数组）

anewarray_quick(0xde)——anewarray字节码的快速版本

概要： Sun公司的JIT编译器中内部使用的anewarray操作码的优化版本。这一操作码不会出现在目前未加载和执行的.class文件中。

初始状态：参见anewarray操作码

最终状态：参见anewarray操作码

areturn(0xb0)——从方法中返回地址结果

概要：从当前方法栈中弹出地址（对象引用），并将这一对象压入到调用者环境中的方法栈。当前方法被终止，控制转移到调用者环境。

初始状态：address

最终状态：(n/a)

arraylength(0xbe)——获得数组长度

概要：从堆栈中弹出数组（地址）并将此数组的长度（integer）压入堆栈。对于多维数组，返回的是第一维的长度。

初始状态：address（数组）引用

最终状态：int

astore<varnum>(0x3a)[byte/short]——存储地址到局部变量中

概要：从堆栈中弹出一个地址（对象引用或子过程的返回位置）并将其存储到第<varnum>个局部变量中。在不使用宽(wide)操作数前缀的情况下，<varnum>的值是一个0...255范围的byte。否则为0...65536范围的short。

初始状态：address

最终状态：-

astore_0(0x4b)——存储地址到局部变量0中

概要：从栈顶弹出一个地址（对象引用）并存储到局部变量0中。此功能等价于astore 0，但它占用的字节数更少且速度更快。

初始状态：address

最终状态：–

astore_1(0x4c)——存储地址到局部变量1中

概要：从栈顶弹出一个地址（对象引用）并存储到局部变量1中。此功能等价于astore 1，但它占用的字节数更少且速度更快。

初始状态：address

最终状态：–

astore_2(0x4d)——存储地址到局部变量2中

概要：从栈顶弹出一个地址（对象引用）并存储到局部变量2中。此功能等价于astore 2，但它占用的字节数更少且速度更快。

初始状态：address

最终状态：–

astore_3(0x2e)———存储地址到局部变量3中

概要：从栈顶弹出一个地址（对象引用）并存储到局部变量3中。此功能等价于astore 3，但它占用的字节数更少且速度更快。

初始状态：address

最终状态：–

athrow(0xbf)——抛出异常/错误

概要：从栈顶弹出一个地址（对象引用）并将此对象作为异常“抛给”预定义的句柄。此对象类型必须为throwable。如果没有预定义的句柄，当前方法会终止，并在调用者环境中重新抛出此异常。这一过程会重复执行，直至找到句柄或没有调用者环境。若无调用者环境，进程/线程将会终止。若找到句柄，对象会压入到句柄的堆栈并将控制转移给句柄。

初始状态：address（throwable）

最终状态：(n/a)

baload(0x33)——从byte数组中加载值

概要：从堆栈中弹出一个数组和integer，取出一维byte数组中指定位置的值。取出的值转换为一个integer并压入栈顶。

初始状态：int（索引）
address(数组引用)

最终状态：int

bastore(0x54)——存储值到byte数组中

概要：将8位的字节存储到byte数组中。顶端所弹出的参数为定义所用位置的索引。第二个弹出的参数为需存储的byte值，第三个（最后一个）参数为数组本身。第二个参数从int截断为byte并存储在数组中。

使用类似的语义，baload也用于从boolean数组中加载值。

初始状态：int（索引）
int(byte 或 boolean)
address（数组引用）

最终状态：–

Bipush<constant>(0x10[byte])——压入integer|byte

概要：作为参数的byte值（−128...127）符号扩展成一个integer后压入栈中。

初始状态：−

最终状态：int

breakpoint(0xca)——断点（保留的操作码）

概要：这一操作码保留为JVM实现内部使用，通常用于支持调试功能。在类文件中出现此操作码将是非法的，这样的类文件也不能通过校验。

初始状态：(n/a)

最终状态：(n/a)

caload(0x34)——从char数组中加载值

概要：从堆栈中弹出一个数组和integer，取出一维char数组中指定位置的值。取出的值转换为一个integer并压入栈顶。

初始状态：int（索引）
　　address(数组引用)

最终状态：int

castore(0x54)——存储值到char数组中

概要：将16位的UTF−16字符存储到char数组中。顶端所弹出的参数为定义数组位置的索引。第二个弹出的参数为需存储的char值，第三个（最后一个）参数为数组本身。第二个参数从int截断为char并存储到数组中。

初始状态：int（索引）
　　int(char)
　　address（数组引用）

最终状态：−

checkcast<type>(0xc0[常数池索引])——确认类型是否兼容

概要：检查（但不弹出）栈顶元素以确认它是否为一个可转换为参数指定类型的地址（对象或数组引用）。换言之，对象或者为null，或者为<type>或<type>父类的一个实例（参见`instanceof`）。字节码中的type为常数池的2字节索引。如果类型不兼容，会抛出Class Cast Exception异常（参见`athrow`）。

初始状态：address

最终状态：address

checkcast_quick(0xe0)——checkcast字节码的快速版本

概要：Sun公司的JIT编译器中内部使用的`checkcast`操作码的优化版本。这一操作码不会出现在目前未加载和执行的`.class`文件中。

初始状态：参见checkcast操作码

最终状态：参见checkcast操作码

d2f(0x90)——将double转换为float

概要：从栈中弹出双字的double，将其转换为单字的float并压入栈中。

初始状态：double
　　double

最终状态：float

d2i(0x8e) ——将double转换为integer

概要：从栈中弹出双字的double，将其转换为单字的integer并压入栈中。

初始状态：double
double

最终状态：int

d2l(0x8f)——将double转换为long

概要：从栈中弹出双字的double，将其转换为双字的long并压入栈中。

初始状态：double
double

最终状态：long
long

dadd(0x63)——double加法

概要：从栈中弹出两个double，计算其和并压入栈中。

初始状态：double−1
double−1
double−2
double−2

最终状态：double
double

daload(0x31)——从double数组中加载值

概要：从堆栈中弹出一个数组和integer，取出一维double数组中指定位置的值并压入栈顶。

初始状态：int（索引）
address(数组引用)

最终状态：double
double

dastore(0x52)——存储值到double数组中

概要：将双字的double存储到double数组中。顶端所弹出的参数为定义所用位置的索引。第二个/三个弹出的参数为需存储的double值，最后一个参数为数组本身。

初始状态：int（索引）
double
double
address（数组引用）

最终状态：−

dcmpg(0x98)——比较double，在存在NaN的情况下返回1

概要：将两个双字的double从操作数栈中弹出，将比较结果−1，0或1（作为一个integer）压入栈中。如果double−2大于double−1，将+1压入栈中，如果二者相等，将0压入栈中，否则的话将−1压入栈中。如果弹出的任意一个字或两个字解释为double时等于IEEE NaN（非数字），则将结果+1压入栈中。

初始状态：double−1
double−1
double−2

double−2

最终状态：int

dcmpl(0x97)——比较double，在存在NaN的情况下返回−1

概要：将两个双字的double从操作数栈中弹出，将比较结果−1，0或1（作为一个integer）压入栈中。如果double−2大于double−1，将+1压入栈中，如果二者相等，将0压入栈中，否则的话将−1压入栈中。如果弹出的任意一个字或两个字解释为double时等于IEEE NaN（非数字），则将结果−1压入栈中。

初始状态：double−1
double−1
double−2
double−2

最终状态：int

dconst_0(0xe)——压入double常量0.0

概要：将64位IEEE双精度浮点数常量0.0压入操作数栈中。

初始状态：−

最终状态：double(0.0)
double(0.0)

dconst_1(0xf)——压入double常量1.0

概要：将64位IEEE双精度浮点数常量1.0压入操作数栈中。

初始状态：−

最终状态：double(1.0)
double(1.0)

ddiv(0x6f)——double除法

概要：弹出两个双字的双精度浮点数，并将double−1除double−2的结果压入栈中。

初始状态：double−1
double−1
double−2
double−2

最终状态：double
double

dload<varnum>(0x18[byte/short])——从局部变量中加载double

概要：从第<varnum>和<varnum>+1个局部变量中加载双字的双精度浮点数并压入堆栈。在不使用宽(`wide`)操作数前缀的情况下，<varnum>的值是一个0...255范围的byte。否则为0...65536范围的short。

初始状态：−

最终状态：double
double

dload_0(0x26)——从局部变量0/1加载double

概要：从局部变量0和1中加载双精度浮点数并压入堆栈中。此功能等价于`dload 0`，但它占用的字节数更少且速度更快。

初始状态：−

最终状态：double

double

dload_1(0x27)——从局部变量1/2加载double

概要：从局部变量1和2中加载双精度浮点数并压入堆栈中。此功能等价于`dload 1`，但它占用的字节数更少且速度更快。

初始状态：—

最终状态：double

double

dload_2(0x28)——从局部变量2/3加载double

概要：从局部变量2和3中加载双精度浮点数并压入堆栈中。此功能等价于`dload 2`，但它占用的字节数更少且速度更快。

初始状态：—

最终状态：double

double

dload_3(0x29)——从局部变量3/4加载double

概要：从局部变量3和4中加载双精度浮点数并压入堆栈中。此功能等价于`dload 3`，但它占用的字节数更少且速度更快。

初始状态：—

最终状态：double

double

dmul(0x6b)——double乘法

概要：弹出两个双字的双精度浮点数，并将二者的乘积压入栈中。

初始状态：double−1

double−1

double−2

double−2

最终状态：double

double

dneg(0x77)——double求反

概要：弹出一个双字的双精度浮点数，符号求反（乘以−1）后压入栈中。

初始状态：double

double

最终状态：double

double

drem(0x73)——double求余

概要：弹出两个双字的双精度浮点数，并将double−1除double−2的余数压入栈中。

初始状态：double−1

double−1

double−2

double−2

最终状态：double

double

dreturn(0xb0)——从方法中返回double

概要：从当前方法栈中弹出双字的双精度浮点数并压入到调用者环境中的方法栈。当前方法被终止，控制转移到调用者环境。

初始状态：double

double

最终状态：(n/a)

dstore<varnum>(0x39)[byte/short]——存储double到局部变量中

概要：从堆栈中弹出一个double并将其存储到第<varnum>和<varnum>+1个局部变量中。在不使用宽(wide)操作数前缀的情况下，<varnum>的值是一个0...255范围的byte。否则为0...65536范围的short。

初始状态：double

double

最终状态：–

dstore_0(0x47)——存储double到局部变量0/1中

概要：从栈顶弹出一个double并存储到局部变量0和1中。此功能等价于`dstore 0`，但它占用的字节数更少且速度更快。

初始状态：double

double

最终状态：–

dstore_1(0x48)——存储地址到局部变量1/2中

概要：从栈顶弹出一个double并存储到局部变量1和2中。此功能等价于`dstore 1`，但它占用的字节数更少且速度更快。

初始状态：double

double

最终状态：–

dstore_2(0x49)——存储double到局部变量2/3中

概要：从栈顶弹出一个double并存储到局部变量2和3中。此功能等价于`dstore 2`，但它占用的字节数更少且速度更快。

初始状态：double

double

最终状态：–

dstore_3(0x4a)——存储double到局部变量3/4中

概要：从栈顶弹出一个double并存储到局部变量3和4中。此功能等价于`astore 3`，但它占用的字节数更少且速度更快。

初始状态：double

double

最终状态：–

dsub(0x67)——double减法

概要：弹出两个双字的双精度浮点数，并将double–2减double–1的结果压入栈中。

初始状态：double–1

double–1

double−2
double−2

最终状态：double
double

dup(0x59)——复制栈顶的字

概要：复制栈顶的一个字并压入栈中。

初始状态：word−1

最终状态：word−1
word−1

dup2(0x5c)——复制栈顶的两个字

概要：复制栈顶的两个字并压入栈中。被复制的项可以是两个独立的单字项（如int、float或地址）或一个双字项（如long或double）。

初始状态：word−1
word−2

最终状态：word−1
word−2
word−1
word−2

dup2_x1(0x5d)——复制栈顶的两个字并插入到第三个字的下面

概要：复制栈顶的两个字并插入到第三个字的下面，作为第四和第五个字。被复制的项可以是两个独立的单字项（如int、float或地址）或一个双字项（如long或double）。

初始状态：word−1
word−2
word−3

最终状态：word−1
word−2
word−3
word−1
word−2

dup2_x2(0x5e)——复制栈顶的两个字并插入到第四个字的下面

概要：复制栈顶的两个字并插入到第四个字的下面，作为第五和第六个字。被复制的项可以是两个独立的单字项（如int、float或地址）或一个双字项（如long或double）。

初始状态：word−1
word−2
word−3
word−4

最终状态：word−1
word−2
word−3
word−4
word−1
word−2

dup_x2(0x5a)——复制栈顶的一个字并插入到第二个字的下面

概要：复制栈顶的一个字并插入到第二个字的下面，作为第三个字。

初始状态：word−1
word−2

最终状态：word−1
word−2
word−1

dup_x2(0x5b)——复制栈顶的一个字并插入到第三个字的下面

概要：复制栈顶的一个字并插入到第三个字的下面，作为第四个字。

初始状态：word−1
word−2

最终状态：word−1
word−2
word−1

f2d(0x8d)——将float转换为double

概要：从栈中弹出单字的float浮点数，将其转换为双字的double并压入栈中。

初始状态：float

最终状态：double
double

f2i(0x8b)——将float转换为int

概要：从栈中弹出单字的float浮点数，将其转换为单字的integer并压入栈中。

初始状态：float

最终状态：int

f2l(0x8c)——将float转换为long

概要：从栈中弹出单字的float浮点数，将其转换为双字的long并压入栈中。

初始状态：float

最终状态：long
long

fadd(0x62)——float加法

概要：从栈中弹出两个float，计算其和并压入栈中。

初始状态：float
float

最终状态：float

faload(0x30)——从float数组中加载值

概要：从堆栈中弹出一个数组和integer，取出一维float数组中指定位置的值并压入栈顶。

初始状态：int（索引）
address(数组引用)

最终状态：float

fastore(0x51)——存储值到float数组中

概要：将单字的float存储到float数组中。顶端所弹出的参数为定义所用位置的索引。第二个弹出的参数为需存储的float值，第三个（最后一个）参数为数组本身。

初始状态：int（索引）

float

address（数组引用）

最终状态：－

fcmpg(0x96)——比较float，在存在NaN的情况下返回1

概要：将两个单字的float从操作数栈中弹出，将比较结果－1，0或1（作为一个integer）压入栈中。如果float－2大于float－1，将+1压入栈中，如果二者相等，将0压入栈中，否则的话将－1压入栈中。如果弹出的任意一个字或两个字解释为浮点数时等于IEEE NaN（非数字），则将+1压入栈中。

初始状态：float－1

float－2

最终状态：int

fcmpl(0x95)——比较float，在存在NaN的情况下返回-1

概要：将两个单字的float从操作数栈中弹出，将比较结果－1，0或1（作为一个integer）压入栈中。如果float－2大于float－1，将+1压入栈中，如果二者相等，将0压入栈中，否则的话将－1压入栈中。如果弹出的两个字或其中的任意一个解释为浮点数时等于IEEE NaN（非数字），则将－1压入栈中。

初始状态：float－1

float－2

最终状态：int

fconst_0(0xb)——压入float常量0.0

概要：将32位IEEE的浮点数常量0.0压入操作数栈中。

初始状态：－

最终状态：float(0.0)

fconst_1(0xc)——压入float常量1.0

概要：将32位IEEE的浮点数常量1.0压入操作数栈中。

初始状态：－

最终状态：float(1.0)

fconst_2(0xd)——压入float常量2.0

概要：将32位IEEE的浮点数常量2.0压入操作数栈中。

初始状态：－

最终状态：float(2.0)

fdiv(0x6e) ——float除法

概要：弹出两个单字的浮点数，并将float－1除float－2的结果压入栈中。

初始状态：float－1

float－1

float－2

float－2

最终状态：float

float

fload<varnum>(0x17[byte/short])——从局部变量中加载float

概要：从第<varnum>个局部变量中加载单字的float浮点数并压入堆栈。在不使用宽(wide)操作数前缀的情况下，<varnum>的值是一个0...255范围的byte。否则为0...65536范围的short。

初始状态：–

最终状态：float

fmul(0x6a)——float乘法

概要：弹出两个单字的float浮点数，并将二者的乘积压入栈中。

初始状态：float
float

最终状态：float
float

fload_0(0x22)——从局部变量0加载float

概要：从局部变量0中加载单字float浮点数并压入堆栈中。此功能等价于`fload 0`，但它占用的字节数更少且速度更快。

初始状态：–

最终状态：float

fload_1(0x23)——从局部变量1加载float

概要：从局部变量1中加载单字float浮点数并压入堆栈中。此功能等价于`fload 1`，但它占用的字节数更少且速度更快。

初始状态：–

最终状态：float

fload_2(0x24)——从局部变量2加载float

概要：从局部变量2中加载单字float浮点数并压入堆栈中。此功能等价于`fload 2`，但它占用的字节数更少且速度更快。

初始状态：–

最终状态：float

fload_3(0x25)——从局部变量3加载float

概要：从局部变量3中加载单字float浮点数并压入堆栈中。此功能等价于`fload 3`，但它占用的字节数更少且速度更快。

初始状态：–

最终状态：float

fneg(0x76)——float求反

概要：弹出一个float浮点数，符号求反（乘以–1）后压入栈中。

初始状态：float

最终状态：float

frem(0x72)——float求余

概要：弹出两个单字的float浮点数，并将float–1除float–2的余数压入栈中。

初始状态：float–1
float–2

最终状态：float

freturn(0xae)——从方法中返回float

概要：从当前方法栈中弹出单字的float浮点数并压入到调用者环境中的方法栈。当前方法被终止，控制转移到调用者环境中。

初始状态：float

最终状态：(n/a)

fstore<varnum>(0x38)[byte/short]——存储float到局部变量中

概要：从堆栈中弹出一个float并将其存储到第<varnum>个局部变量中。在不使用宽(wide)操作数前缀的情况下，<varnum>的值是一个0...255范围的byte。否则为0...65536范围的short。

初始状态：float

最终状态：—

fstore_0(0x43)——存储float到局部变量0中

概要：从栈顶弹出一个float并存储到局部变量0中。此功能等价于`fstore 0`，但它占用的字节数更少且速度更快。

初始状态：float

最终状态：—

fstore_1(0x44)——存储float到局部变量1中

概要：从栈顶弹出一个float并存储到局部变量1中。此功能等价于`fstore 1`，但它占用的字节数更少且速度更快。

初始状态：float

最终状态：—

fstore_2(0x45)——存储float到局部变量2中

概要：从栈顶弹出一个float并存储到局部变量2中。此功能等价于`fstore 2`，但它占用的字节数更少且速度更快。

初始状态：float

最终状态：—

fstore_3(0x46)——存储float到局部变量3中

概要：从栈顶弹出一个float并存储到局部变量#3中。此功能等价于`fstore 3`，但它占用的字节数更少且速度更快。

初始状态：float

最终状态：—

fsub(0x67)——float减法

概要：弹出两个单字的float浮点数，并将float−2减float−1的结果压入栈中。

初始状态：float−1

float−2

最终状态：float

getfield<fieldname><type>(0xb4[short][short])——获得对象域

概要：从堆栈中弹出一个地址（对象引用），获得指定域值并压入栈中。`getfield`操作码带有两个参数，域标识符和域类型，它们作为2个字节的常数池索引存储在字节码中。不同于Java的是，域名必须完全限定，包括相关的类和包名。

初始状态：地址（对象）

最终状态：值

getfield2_quick(0xd0)——用于双字域的getfield的快速版本

概要：Sun公司的JIT编译器中内部使用的getfield操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见getfield操作码

最终状态：参见getfield操作码

getfield_quick(0xce)——getfield操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的getfield操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见getfield操作码

最终状态：参见getfield操作码

getfield_quick_w(0xe3)——getfield的快速宽版本

概要：Sun公司的JIT编译器中内部使用的getfield操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见getfield操作码

最终状态：参见getfield操作码

getstatic<fieldname><type>(0xb2[short][short])——获得类域

概要：获得指定类域的值并压入栈中。getstatic操作码带有两个参数，域标识符和域类型，它们作为2个字节的常数池索引存储在字节码中。不同于Java的是，域名必须完全限定，包括相关的类和包名。

初始状态：－

最终状态：值

getstatic_quick(0xd2)——getstatic操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的getstatic操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见getstatic操作码

最终状态：参见getstatic操作码

getstatic2_quick(0xd4)——用于2字节getstatic操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的getstatic操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见getstatic操作码

最终状态：参见getstatic操作码

goto<label>(0xa7[short])——无条件跳转到label处

概要：无条件地将控制转移到<label>标识的位置。在指令字节码中，操作码的后面是相对于当前PC值的两个字节的偏移量。如果标识的偏移量大于两个字节的表示，可以使用goto_w。jasmin汇编器能够基于偏移量的分析决定使用哪个操作码。

初始状态：－

最终状态：－

goto_w<label>(0xa7[short])——使用宽偏移量无条件跳转到label处

概要：无条件地将控制转移到<label>标识的位置。在指令字节码中，操作码的后面是相对于当前PC值的四个字节的偏移量。使用此操作码，可以跳转到超过32 767字节的偏移量处。

jasmin汇编器能够基于偏移量的分析自动决定使用goto还是goto_w操作码。

初始状态：－

最终状态：－

i2b(0x91)——将integer转换为byte

概要：从栈中弹出单字的integer，将其截断为一个byte(0...255),高位填0扩展为32bit，并压入栈中（作为一个integer）。

初始状态：int

最终状态：int

i2c(0x87)——将integer转换为char

概要：从栈中弹出单字的integer，将其截断为一个两字节的UTF－16 char，高位填0扩展为32bit，并压入栈中（作为一个integer）。

初始状态：int

最终状态：int

i2d(0x87)——将integer转换为double

概要：从栈中弹出单字的integer，将其转换为双字的double并压入栈中。

初始状态：integer

最终状态：double
double

i2f(0x86)——将integer转换为float

概要：从栈中弹出单字的integer，将其转换为单字的浮点数并压入栈中。

初始状态：integer

最终状态：float

i2l(0x85)——将integer转换为long

概要：从栈中弹出单字的integer，将其符号扩展为一个双字的long并压入栈中。

初始状态：int

最终状态：long
long

i2s(0x93)——将integer转换为short

概要：从栈中弹出单字的integer，将其截断为一个有符号的short(－32768...32767)并符号扩展为32bit，压入栈中（作为一个integer）。

初始状态：int

最终状态：int

iadd(0x60)——integer加法

概要：从栈中弹出两个integer，计算其和并压入栈中。

初始状态：int
int

最终状态：int

iaload(0x2e)——从integer数组中加载值

概要：从堆栈中弹出一个数组和integer，取出一维integer数组中指定位置的值并压入栈顶。

初始状态：int（索引）

address(数组引用)

最终状态：int

iand(0x7e)——integer逻辑与

概要：从栈中弹出两个integer，计算位与的结果，并将32-bit的integer结果压入栈中。

初始状态：int

int

最终状态：int

iastore(0x4f)——存储值到integer数组中

概要：将单字的integer存储到integer数组中。顶端所弹出的参数为定义所用位置的索引。第二个弹出的参数为需存储的integer值，第三个（最后一个）参数为数组本身。

初始状态：int（索引）

int（值）

address（数组引用）

最终状态：–

iconst_0(0x03)——压入integer常量0

概要：将32-bit 的integer常量0（0x0）压入操作数栈中。

初始状态：–

最终状态：int(0)

iconst_1(0x4)——压入integer常量1

概要：将32-bit 的integer常量1（0x1）压入操作数栈中。

初始状态：–

最终状态：int(1)

iconst_2(0x5)——压入integer常量2

概要：将32-bit 的integer常量2（0x2）压入操作数栈中。

初始状态：–

最终状态：int(2)

iconst_3(0x6)——压入integer常量3

概要：将32-bit 的integer常量3（0x3）压入操作数栈中。

初始状态：–

最终状态：int(3)

iconst_4(0x7)——压入integer常量4

概要：将32-bit 的integer常量4（0x4）压入操作数栈中。

初始状态：–

最终状态：int(4)

iconst_5(0x8)——压入integer常量5

概要：将32-bit 的integer常量5（0x5）压入操作数栈中。

初始状态：–

最终状态：int(5)

iconst_m1(0x2)——压入integer常量–1

概要：将32-bit 的integer常量–1（0xFFFF）压入操作数栈中。

初始状态：–

最终状态：int(−1)

idiv(0x6c)——integer除法

概要：弹出两个单字的integer浮点数，并将integer−1除integer−2的结果压入栈中。

初始状态：int−1

int−2

最终状态：int

if_acmpeq<label>(0xa5[short])——比较地址，相等则跳转

概要：将两个地址（对象引用）从操作数栈中弹出，如果二者相等，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：address

address

最终状态：−

if_acmpne<label>(0xa6[short])——比较地址，不相等则跳转

概要：将两个地址（对象引用）从操作数栈中弹出，如果二者不相等，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：address

address

最终状态：−

if_icmpeq<label>(0x9f[short])——比较integer，相等则跳转

概要：将两个integer从操作数栈中弹出，如果二者相等，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：int

int

最终状态：−

if_icmpge<label>(0xa2[short])——比较integer，大于等于则跳转

概要：将两个integer从操作数栈中弹出，如果倒数第二个元素大于等于栈顶元素，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：int

int

最终状态：−

if_icmpgt<label>(0xa3[short])——比较integer，大于则跳转

概要：将两个integer从操作数栈中弹出，如果倒数第二个元素大于栈顶元素，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：int

int

最终状态：−

if_icmple<label>(0xa2[short])——比较integer，小于等于则跳转

概要：将两个integer从操作数栈中弹出，如果倒数第二个元素小于等于栈顶元素，控制

转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：int

int

最终状态：－

if_icmplt<label>(0xa1[short])——比较integer，小于则跳转

概要：将两个integer从操作数栈中弹出，如果倒数第二个元素小于栈顶元素，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：int

int

最终状态：－

if_icmpne<label>(0xa0[short])——比较integer，不相等则跳转

概要：将两个integer从操作数栈中弹出，如果二者不相等，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：int

int

最终状态：－

ifeq<label>(0x99[short])——相等则跳转

概要：将一个integer从操作数栈中弹出，如果等于0，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：int

最终状态：－

ifge<label>(0x9c[short])——大于等于则跳转

概要：将一个integer从操作数栈中弹出，如果大于等于0，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：int

最终状态：－

if_gt<label>(0x9d[short])——大于则跳转

概要：将一个integer从操作数栈中弹出，如果大于0，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：int

最终状态：－

if_le<label>(0x9e[short])——小于等于则跳转

概要：将一个integer从操作数栈中弹出，如果小于等于0，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：int

最终状态：－

if_lt<label>(0x9b[short])——小于则跳转

概要：将一个integer从操作数栈中弹出，如果小于0，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：int

最终状态：–

if_ne<label>(0x9a[short])——不相等则跳转

概要：将一个integer从操作数栈中弹出，如果不等于0，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：int

最终状态：–

if_nonnull<label>(0xc7[short])——不为null则跳转

概要：将一个地址（对象引用）从操作数栈中弹出，如果不为null，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：address

最终状态：–

ifnull<label>(0xc6[short])——为null则跳转

概要：将一个地址（对象引用）从操作数栈中弹出，如果为null，控制转移到<label>处。在内部，操作码后为2个字节的偏移量。当产生跳转时，偏移量会加到当前的PC值上。

初始状态：address

最终状态：–

iinc<varnum> <increment> (0x84[byte/short] [byte/short])——递增局部变量中的integer

概要：将包含一个增量的局部变量递增。第一个参数定义要调整的局部变量编号，第二个参数为所调整的有符号常量。第一个参数的范围为0...255，第二个参数的范围为–128...127。如果使用`wide`前缀，第一个参数的范围为0...65536，第二个参数的范围为–32768...32767。堆栈不发生变化。

初始状态：–

最终状态：–

iload<varnum>(0x15[byte/short])——从局部变量中加载integer

概要：从第<varnum>个局部变量中加载单字的integer并压入堆栈。在不使用宽(`wide`)操作数前缀的情况下，<varnum>的值是一个0...255范围的byte。否则为0...65536范围的short。

初始状态：–

最终状态：int

iload_0(0x1a)——从局部变量0加载integer

概要：从局部变量0中加载单字integer并压入堆栈中。此功能等价于`iload 0`，但它占用的字节数更少且速度更快。

初始状态：–

最终状态：int

iload_1(0x1b)——从局部变量1加载integer

概要：从局部变量1中加载单字integer并压入堆栈中。此功能等价于`iload 1`，但它占用的字节数更少且速度更快。

初始状态：–

最终状态：int

iload_2(0x1c)——从局部变量2加载integer

概要：从局部变量2中加载单字integer并压入堆栈中。此功能等价于iload 2，但它占用的字节数更少且速度更快。

初始状态：–

最终状态：int

iload_3(0x1d)——从局部变量3加载integer

概要：从局部变量3中加载单字integer并压入堆栈中。此功能等价于iload 3，但它占用的字节数更少且速度更快。

初始状态：–

最终状态：int

imdep1(0xfe)——保留的操作码

概要：这一操作码保留为JVM实现的内部使用。这一操作码出现在.class文件中是非法的并且这样的类文件会校验失败。

初始状态：(n/a)

最终状态：(n/a)

imdep2(0xff)——保留的操作码

概要：这一操作码保留为JVM实现的内部使用。这一操作码出现在.class文件中是非法的并且这样的类文件会校验失败。

初始状态：(n/a)

最终状态：(n/a)

imul(0x68)——integer乘法

概要：弹出两个integer，并将二者的乘积压入栈中。

初始状态：int
int

最终状态：int

ineg(0x74)——integer求反

概要：弹出一个integer，符号求反（乘以–1）后压入栈中。

初始状态：int

最终状态：int

instanceof<type>(0xc1[short])——检测对象/数组是否为指定类型

概要：从栈中弹出一个地址（对象或数组引用）并判断是否同指定的type兼容–或者为type的实例，或者实现了那一接口，或者为type父类的实例。如果兼容，压入整数1，否则压入0。

初始状态：address

最终状态：int

instanceof_quick(0xe1)——instanceof操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的instanceof操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见instanceof操作码

最终状态：参见instanceof操作码

invokeinterface<method><Nargs>(0xb9[short] [byte] [byte])——调用接口方法

概要：调用接口（相对于类）中定义的方法。invokeinterface的参数包括被调用方法的完全限定名（包括接口名、参数类型和返回类型）和参数个数。这些参数和实现接口的对象地址（对象引用）一起从栈中弹出。一个新的栈帧会为被调用环境创建，同时对象和参数也会压入环境栈中，控制转移到新的方法/环境。在方法返回时，返回值（由return给出）会压入到调用者环境栈中。

在字节码中，方法名为两个字节的常数池（参见词汇表）索引。下一个字节为参数的个数（以字为单位），最多可达255个。在下一个字节存储值0，在JVM内部可用来存储哈希值以加快方法的查找速度。

初始状态：argN

...

arg2

arg1

address(对象)

最终状态：(结果)

invokeinterface_quick(0xda)——invokeinterface操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的invokeinterface操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见invokeinterface操作码

最终状态：参见invokeinterface操作码

invokenonvirtual_quick(0xda)——invokespecial操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的invokespecial操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见invokespecial操作码

最终状态：参见invokespecial操作码

invokespecial<method> (0xb7[short])——调用实例方法

概要：在下述情况下使用invokespecial调用对象中的实例方法：

- 实例初始化方法<init>
- this的私有方法
- this父类中的方法

这一操作码类似于invokevirtual。invokespecial的参数包括调用方法的完全限定名（包括接口名、参数类型和返回类型）和参数个数。这些参数和相关类的实例地址（对象引用）一起从栈中弹出。一个新的栈帧会为被调用环境创建，同时对象和参数也会压入环境栈中，控制转移到新的方法/环境。在方法返回时，返回值（由return给出）会压入到调用者环境栈中。在字节码中，方法名为两个字节的常数池（参见词汇表）索引。

初始状态：argN

...

arg2

arg1

address（对象）

最终状态：(结果)

invokestatic<method> (0xb8[short])——调用静态方法

概要：调用类的静态方法。invokestatic的参数包括调用方法的完全限定名（包括接口名、参数类型和返回类型）和参数个数。这些参数从栈中弹出。一个新的栈帧会为被调用环境创建，同时对象和参数也会压入环境栈中，控制转移到新的方法/环境。在方法返回时，返回值（由return给出）会压入到调用者环境栈中。在字节码中，方法名为两个字节的常数池（参见词汇表）索引。

初始状态：argN

...

arg2

arg1

最终状态：(结果)

invokestatic_quick(0xd9)——invokestatic操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的invokestatic操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见invokestatic操作码

最终状态：参见invokestatic操作码

invokesuper_quick(0xd8)——invokespecial操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的invokespecial操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见invokespecial操作码

最终状态：参见invokespecial操作码

invokevirtual<method> (0xb6[short])——调用实例方法

概要：调用对象的实例方法。invokevirtual的参数包括调用方法的完全标识名（包括接口名、参数类型和返回类型）和参数个数。这些参数和相关类的实例地址（对象引用）一起从栈中弹出。一个新的栈帧会为被调用环境创建，同时对象和参数也会压入环境栈中，控制转移到新的方法/环境。在方法返回时，返回值（由return给出）会压入到调用者环境栈中。在字节码中，方法名为两个字节的常数池（参见词汇表）索引。

初始状态：argN

...

arg2

arg1

address（对象）

最终状态：(结果)

invokevirtual_quick(0xd6)——invokevirtual操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的invokevirtual操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见invokevirtual操作码

最终状态：参见invokevirtual操作码

invokevirtual_quick_w(0xe2)——invokevirtual操作码的快速版本（宽索引）

概要：Sun公司的JIT编译器中内部使用的invokevirtual操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见invokevirtual操作码

最终状态：参见invokevirtual操作码

invokevirtualobject_quick(0xd6) ——invokevirtual调用对象方法的快速版本

概要：Sun公司的JIT编译器中内部使用的invokevirtual操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见invokevirtual操作码

最终状态：参见invokevirtual操作码

ior(0x80)——integer逻辑或

概要：从栈中弹出两个integer，计算它们的位或结果并作为32-bit的整数压入栈中。

初始状态：int
int

最终状态：int

irem(0x70)——integer求余

概要：弹出两个单字的integer，并将int-1除int-2的余数压入栈中。这一操作类似于C或Java的%运算。

初始状态：int-1
int-2

最终状态：int

ireturn(0xac)——从方法中返回integer

概要：从当前方法栈中弹出一个integer并压入到调用者环境中的方法栈。当前方法被终止，控制转移到调用者环境中。

初始状态：int

最终状态：(n/a)

ishl(0x78)——integer左移

概要：弹出两个integer，将栈顶下的第二个元素左移栈顶元素低6位所示的次数，并将结果压入栈中。新空出的位置用0填充。这相当于乘2操作，但速度会更快。

初始状态：int（移位）
int（值）

最终状态：int

ishr(0x7a)——integer右移

概要：弹出两个integer，将栈顶下的第二个元素右移栈顶元素低6位所示的次数，并将结果压入栈中。注意：这是一个算数移位，符号位将会填充到新空出的位置中。

初始状态：int（移位）
int（值）

最终状态：int

istore<varnum>(0x36)[byte/short] ——存储integer到局部变量中

概要：从堆栈中弹出一个integer并将其存储到第<varnum>个局部变量中。在不使用宽(wide)操作数前缀的情况下，<varnum>的值是一个0...255范围的byte。否则为0...65536范围的short。

初始状态：int

最终状态：–

istore_0(0x3b)——存储integer到局部变量0中

概要：从栈顶弹出一个integer并存储到局部变量0中。此功能等价于`istore 0`，但它占用的字节数更少且速度更快。

初始状态：int

最终状态：–

istore_1(0x3c)——存储integer到局部变量1中

概要：从栈顶弹出一个integer并存储到局部变量1中。此功能等价于`istore 1`，但它占用的字节数更少且速度更快。

初始状态：int

最终状态：–

istore_2(0x3d)——存储integer到局部变量2中

概要：从栈顶弹出一个integer并存储到局部变量2中。此功能等价于`istore 2`，但它占用的字节数更少且速度更快。

初始状态：integer

最终状态：–

istore_3(0x3e)——存储integer到局部变量3中

概要：从栈顶弹出一个integer并存储到局部变量3中。此功能等价于`istore 3`，但它占用的字节数更少且速度更快。

初始状态：int

最终状态：–

isub(0x64)——integer减法

概要：弹出两个integer，计算栈顶下的第二个元素和栈顶元素的差值并压入栈中。

初始状态：int
int

最终状态：int

iushr(0x7c)——无符号int右移

概要：弹出两个integer，将栈顶下的第二个元素右移栈顶元素低6位所示的次数，并将结果压入栈中。注意：这是一个逻辑移位，符号位被忽略，0将会填充到新空出的位置中。

初始状态：int（移位）
int（值）

最终状态：int

ixor(0x82)——integer逻辑异或

概要：从栈中弹出两个integer，计算它们的位异或结果并作为32-bit的整数压入栈中。

初始状态：int
int

最终状态：int

jsr_w<label>(0xc9 [int])——使用宽偏移量跳转到子过程

概要：将下一指令地址（PC+5，`jsr_w`的指令长度为5）压入栈中，并无条件跳转到<label>处。

初始状态：–

最终状态：address(locn)

jsr_w<label>(0xa8 [short])——跳转到子过程

概要：将下一指令地址（PC+3，jsr的指令长度为3）压入栈中，并无条件跳转到<label>处。

初始状态：–

最终状态：address(locn)

l2f(0x89)——将long转换为float

概要：从栈中弹出双字的long，将其转换为一个单字的float并压入栈中。

初始状态：long
long

最终状态：float

l2i(0x88)——将long转换为int

概要：从栈中弹出双字的long，将其转换为一个单字的integer并压入栈中。注意：由于long的原始符号位丢失，这可能会导致符号位的变化。

初始状态：long
long

最终状态：int

ladd(0x61)——long加法

概要：弹出两个long，计算其和并压入栈中。

初始状态：long-1
long-1
long-2
long-2

最终状态：long
long

laload(0x2f)——从long数组中加载值

概要：从堆栈中弹出一个数组和integer，取出一维long数组中指定位置的值并压入栈顶。

初始状态：int（索引）
address(数组引用)

最终状态：long
long

land(0x7f)——long逻辑与

概要：从栈中弹出两个long，计算位与的结果，并将64–bit的long结果压入栈中。

初始状态：long-1
long-1
long-2
long-2

最终状态：long
long

lastore(0x50)——存储值到long数组中

概要：将双字的long存储到long数组中。顶端所弹出的参数为定义所用位置的索引。第二/三个弹出的参数为需存储的long值，最后一个参数为数组本身。

初始状态：int（索引）

long
long
address（数组引用）

最终状态：－

lcmp(0x94)——比较long

概要：将两个双字的long从操作数栈中弹出并比较。如果long−2大于long−1，将+1压入栈中，如果二者相等，将0压入栈中，否则的话将−1压入栈中。

初始状态：long-1
long-1
long-2
long-2

最终状态：int

lconst_0(0x9)——压入long常量0

概要：将64位的long常量0压入操作数栈中。

初始状态：－

最终状态：long(0)
long(0)

lconst_1(0xa)——压入long常量1

概要：将64位的long常量1压入操作数栈中。

初始状态：－

最终状态：long(1)
long(1)

ldc<constant>(0x12 [short])——加载单字常量

概要：从常数池（参见词汇表）中加载单字值并压入栈中。<constant>可为int、float或字符串，它存储在常数池索引0...255的项中。

初始状态：－

最终状态：word

ldc2_w<constant>(0x14 [int])——加载双字常量

概要：从常数池（参见词汇表）中加载双字值并压入栈中。<constant>可为double或long，它存储在常数池索引0...65536的项中。

初始状态：－

最终状态：word

ldc_quick(0xcb)——ldc操作码的快速版本

概要： Sun公司的JIT编译器中内部使用的ldc操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见ldc操作码

最终状态：参见ldc操作码

ldc_w<constant>(0x13 [int])——使用宽索引加载单字常量

概要：从常数池（参见词汇表）中加载单字值并压入栈中。<constant>可为int、float或字符串，它存储在常数池索引0...65536的项中。

初始状态：－

最终状态：word

ldc_w_quick(0xcd)——ldc_w操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的ldc_w操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见ldc_w操作码

最终状态：参见ldc_w操作码

ldiv(0x6d)——long除法

概要：弹出两个双字的long，并将long−1除long−2的结果压入栈中。

初始状态：long-1
long-1
long-2
long-2

最终状态：long
long

lload<varnum>(0x16[byte/short])——从局部变量中加载long

概要：从第<varnum>和<varnum+1>个局部变量中加载双字的long并压入堆栈。在不使用宽(wide)操作数前缀的情况下，<varnum>的值是一个0...255范围的byte。否则为0...65536范围的short。

初始状态：－

最终状态：long
long

lload_0(0x1e)——从局部变量0和1加载long

概要：从局部变量0和1中加载双字long并压入堆栈中。此功能等价于lload 0，但它占用的字节数更少且速度更快。

初始状态：－

最终状态：long
long

lload_1(0x1f)——从局部变量1和2加载long

概要：从局部变量1和2中加载双字long并压入堆栈中。此功能等价于lload 1，但它占用的字节数更少且速度更快。

初始状态：－

最终状态：long
long

lload_2(0x20)——从局部变量2和3加载long

概要：从局部变量2和3中加载双字long并压入堆栈中。此功能等价于lload 2，但它占用的字节数更少且速度更快。

初始状态：－

最终状态：long
long

lload_3(0x21)——从局部变量3和4加载long

概要：从局部变量3和4中加载双字long并压入堆栈中。此功能等价于lload 3，但它占用

的字节数更少且速度更快。

初始状态：—

最终状态：long

long

lmul(0x69)——long乘法

概要：弹出两个long，并将二者的乘积压入栈中。

初始状态：long-1

long-1

long-2

long-2

最终状态：long

long

lneg(0x75) ——long求反

概要：弹出一个long，符号求反（乘以−1）后压入栈中。

初始状态：long

最终状态：long

lookupswtich<args>(0xab [args])——多路跳转

概要：类似于Java/C++中的switch语句，执行一个多路跳转。弹出栈顶的integer并同一组value:label对相比较。如果integer的值等于value的话，控制转移到label处。如果没有匹配的值，控制转移到缺省的label处。label的实现采用的是相对偏移量，把它的值加到当前的PC上，得到要执行指令的地址。图B-1所示的是使用lookupswitch语句的示例。lookupswitch语句的参数个数不定，因此字节码的存储上有一点复杂。在操作码（0xab）后面，有0到3个填充字节，以使得4字节的缺省偏移量的开始位置是4字节的整数倍。接下来的4个字节定义的是value:label对的个数，每一对按照升序排列连续存储，其中包括4字节的integer和4字节的偏移量，结构如表B-1所示。

```
lookupswitch
    1 : One
    2 : Two
    3 : Three
    5 : Five
  default:Elsewhere
```

图B-1 lookupswitch示例

初始状态：int

最终状态：—

表B-1 lookupswitch字节码的结构

操作码（0xab）和填充
缺省偏移量
条目的个数
value 1 offset 1
value 2 offset 2 …
value N offset N



lor(0x80)——long逻辑或

概要：从栈中弹出两个long，计算它们的位或结果并作为64-bit的long压入栈中。

初始状态：long-1
long-1
long-2
long-2

最终状态：long
long

lrem(0x70)——long求余

概要：弹出两个单字的integer，并将long-1除long-2的余数压入栈中。这一操作类似于C或Java的%运算。

初始状态：long-1
long-1
long-2
long-2

最终状态：long
long

lreturn(0xad)——从方法中返回long

概要：从当前方法栈中弹出一个双字的long并压入到调用者环境中的方法栈。当前方法被终止，控制转移到调用者环境中。

初始状态：long
long

最终状态：(n/a)

lshl(0x79)——long左移

概要：从栈中弹出一个integer和64-bit的long，将long左移integer低6位所示的次数，并将结果压入栈中。新空出的位置用0填充。这相当于乘2操作，但速度会更快。

初始状态：int（移位）
long
long

最终状态：long
long

lshr(0x7b)——long右移

概要：从栈中弹出一个integer和64-bit的long，将long右移integer低6位所示的次数，并将结果压入栈中。新空出的位置用0填充。这相当于乘2操作，但速度会更快。注意：这是一个算术移位，符号位将会填充到新空出的位置中。

初始状态：int（移位）
long
long

最终状态：long
long

lstore<varnum>(0x37)[byte/short]——存储long到局部变量中

概要：从堆栈中弹出一个long并将其存储到第#<varnum>和#<varnum+1>个局部变量中。在不使用宽(wide)操作数前缀的情况下，<varnum>的值是一个0...255范围的byte。否则为0...65536范围的short。

初始状态：long
　　　　　long

最终状态：–

lstore_0(0x3f)——存储long到局部变量0和1中

概要：从栈顶弹出一个long并存储到局部变量0和1中。此功能等价于lstore 0，但它占用的字节数更少且速度更快。

初始状态：long
　　　　　long

最终状态：–

lstore_1(0x40)——存储long到局部变量1和2中

概要：从栈顶弹出一个long并存储到局部变量1和2中。此功能等价于lstore 1，但它占用的字节数更少且速度更快。

初始状态：int

最终状态：–

lstore_2(0x41)——存储long到局部变量2和3中

概要：从栈顶弹出一个long并存储到局部变量2和3中。此功能等价于lstore 2，但它占用的字节数更少且速度更快。

初始状态：long
　　　　　long

最终状态：–

lstore_3(0x42)——存储long到局部变量3和4中

概要：从栈顶弹出一个long并存储到局部变量3和4中。此功能等价于lstore 3，但它占用的字节数更少且速度更快。

初始状态：long
　　　　　long

最终状态：–

lsub(0x65)——long减法

概要：弹出两个双字的long，计算栈顶下的第二个元素和栈顶元素的差值并压入栈中。

初始状态：long-1
　　　　　long-1
　　　　　long-2
　　　　　long-2

最终状态：long
　　　　　long

lushr(0x7d)——无符号long右移

概要：弹出一个integer和一个64-bit的long，将long右移integer低6位所示的次数，并将结果压入栈中。注意：这是一个逻辑移位，符号位被忽略，0将会填充到新空出的位置中。

PDG

初始状态：int（移位）
long
long
最终状态：long
long

lxor(0x83)——long逻辑异或

概要：从栈中弹出两个long，计算它们的位异或结果并作为64-bit的long压入栈中。
初始状态：long-1
long-1
long-2
long-2
最终状态：long
long

monitorenter(0x7d)——获得对象锁

概要：JVM monitor系统能够协调并同步多线程对对象的访问。monitorenter从栈中弹出一个地址（对象引用）并从JVM请求一个独占的对象锁。如果没有其他线程锁定该对象，则发放该对象锁并继续执行。否则的话线程被阻塞并停止执行，直至其他线程通过monitorexit释放对象锁。
初始状态：address
最终状态：–

monitorexit(0xc3——释放对象锁

概要：从栈中弹出一个地址（对象引用）并释放获得的（通过monitorenter）对象锁，以使得其他线程能够获得对象锁。
初始状态：address
最终状态：–

multianewarray<type><N>(0xc5[short][byte])——创建多维数组

概要：为类型为<type>的N维数组分配空间并将数组引用压入栈中。字节码中存储的类型为2字节的常数池（参见词汇表）索引，维数N为0...255的一个字节。操作码的执行会从栈中弹出N个integer，代表数组每一维的大小。创建的数组实际上为一个（子）数组的数组。
初始状态：维数 N
维数 N−1
...
维数1
最终状态：address

multianewarray_quick(0xdf)——multianewarray操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的multianewarray操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。
初始状态：参见multianewarray操作码
最终状态：参见multianewarray操作码

new<class>(0xbb[short])——创建新对象

概要：创建指定类的新对象。字节码中存储的类型为2字节的常数池（参见词汇表）索引。

初始状态：–

最终状态：address（对象）

new_quick(0xdd)——new操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的new操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见new操作码

最终状态：参见new操作码

newarray<typename>(0xbc[type-byte])——创建一维数组

概要：为类型为<typename>的一维数组分配空间并将数组引用压入栈中。字节码中存储的类型为2字节的常数池（参见词汇表）索引，新数组的大小从栈中弹出，数组的类型由操作码后的一个字节表示，其具体含义如下：

boolean	4	byte	8
char	5	short	9
float	6	int	10
double	7	long	11

初始状态：int（大小）

最终状态：address（数组）

nop(0x0)—— 空操作

概要：空操作。空操作主要用于时序调整、调试和将来代码的占位符。

初始状态：–

最终状态：–

pop(0x57)——从栈中弹出单字

概要：弹出并丢弃掉栈顶的字（一个integer、float或地址）。注意：没有相对应的push指令，因为压栈操作是与类型相关联的，如：sipush或ldc。

初始状态：word

最终状态：–

pop2(0x58)——从栈中弹出双字

概要：弹出并丢弃掉栈顶的双字（可以是两个单字量如integer、float或地址，或一个双字量如long或double）。注意：没有相对应的push指令，因为压栈操作是与类型相关联的，如：sipush或ldc2_w。

初始状态：word
word

最终状态：–

putfield<fieldname><type>(0xb5[short] [short])——对象域赋值

概要：从栈中弹出一个地址（对象引用）和值，并将值存储到指定的对象域中。putfield操作码带有两个参数，域标识符和域类型，它们在字节码中存储为2字节的常数池（参见词汇表）索引。不同于Java的是，域名必须是完全限定名，包括相关的类和包名。

初始状态：值
address（对象）

最终状态：–

putfield2_quick(0xd1)——用于2字节putfield操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的putfield操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见putfield操作码

最终状态：参见putfield操作码

putfield_quick(0xcf)——putfield操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的putfield操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见putfield操作码

最终状态：参见putfield操作码

putfield_quick_w(0xe4)——宽索引putfield操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的putfield操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见putfield操作码

最终状态：参见putfield操作码

putstatic<fieldname><type>(0xb3[short] [short])——类域赋值

概要：从栈中弹出一个地址（对象引用）和值，并将值存储到指定的类域中。putstatic操作码带有两个参数，域标识符和域类型，它们在字节码中存储为2字节的常数池（参见词汇表）索引。不同于Java的是，域名必须是完全限定名，包括相关的类和包名。

初始状态：值

最终状态：－

putstatic2_quick(0xd5)——putstatic操作码的替代快速版本

概要：Sun公司的JIT编译器中内部使用的putstatic操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见putstatic操作码

最终状态：参见putstatic操作码

putstatic_quick(0xd3)——putstatic操作码的快速版本

概要：Sun公司的JIT编译器中内部使用的putstatic操作码的优化版本。这一操作码不会在目前未加载和执行的.class文件中出现。

初始状态：参见putstatic操作码

最终状态：参见putstatic操作码

ret<varnum>(0xa9 [byte/short])——从子过程返回

概要：在通过jsr或jsr_w跳转到子过程后，返回到<varnum>局部变量中存储的地址。在不使用宽(wide)操作数前缀的情况下，<varnum>的值是一个0...255范围的byte。否则为0...65536范围的双字节量。

初始状态：－

最终状态：－

return(0xb1)——不带返回结果从方法返回

概要：结束当前方法并将控制转移回调用环境。

初始状态：—

最终状态：(n/a)

saload(0x56)——从short数组中加载值

概要：从堆栈中弹出一个数组和integer，取出一维short数组中指定位置的16-bit的short值，符号扩展成一个integer并压入栈顶。

初始状态：int（索引）
　　　　　address(数组引用)

最终状态：int

sastore(0x56)——存储值到short数组中

概要：将16-bit的short存储到short数组中。顶端所弹出的参数为定义所用位置的索引。第二个弹出的参数为需存储的short值，第三个（最后一个）参数为数组本身。第二个参数由int截断为short并存储在数组中。

初始状态：int（索引）
　　　　　int（short）
　　　　　address（数组引用）

最终状态：—

sipush<constant>(0x11 [short])—— 将[integer]short压入栈中

概要：参数short的值(-32768...32767)符号扩展为一个integer并压入栈中。

初始状态：—

最终状态：int

swap(0x5f)——交换栈顶的两个元素

概要：交换栈顶的两个单字的元素。遗憾的是不存在`swap2`指令。

初始状态：word-1
　　　　　word-2

最终状态：word-2
　　　　　word-1

tableswtich<args>(0xaa [args])——多路跳转

概要：类似于Java/C++中的switch语句，执行一个多路跳转。弹出栈顶的integer并同一组value:label对相比较。如果integer的值等于value的话，控制转移到label处。如果没有匹配的值，控制转移到缺省的label处。label的实现采用的是相对偏移量，把它的值加到当前的PC上，得到要执行指令的地址。这一指令的执行比`lookupswitch`效率更高，但要求值必须是顺序且连续的。

tableswitch语句的表中包括最小和最大值。如果从栈中弹出的integer小于最小值或大于最大值，则控制转移到缺省的label处，否则，控制直接转移（不需要比较）到表中弹出的值减掉最小值的label处。

图B-2所示的是使用的`tableswitch`语句的示例。

`tableswitch`语句的参数个数不定，因此字节码的存储上有一点复杂。在操作码（0xab）后面，有0到3个填充字节，以使得4字节的缺省偏移量的开始位置是4字节的整数倍。接下来的8个字节表示表中的最小值和最大值，接下来的偏移量顺序存储，每个包括4个字节。结构如表B-2所示。

初始状态：int

最终状态：—

```
tableswitch 1 3
    One
    Two
    Three
   default:Elsewhere
```

图B-2 tableswitch示例

表B-2 tableswitch字节码的结构

操作码（0xaa）和填充
缺省偏移量
低值
高值
offset 1
offset 2 …
offset N

wide(0xc4)——指定下一个操作码为宽索引

概要：这是一个操作码前缀而非操作码。它表示下一操作的参数有可能比正常情况要大一些。如：在iload中使用大于255的局部变量。jasmin汇编器在需要时会自动生成这一前缀。

初始状态：—

最终状态：—



附录C 按序号排列的操作代码

C.1 标准操作代码

0	0x0	nop	46	0x2e	iaload
1	0x1	aconst_null	47	0x2f	laload
2	0x2	iconst_m1	48	0x30	faload
3	0x3	iconst_0	49	0x31	daload
4	0x4	iconst_1	50	0x32	aaload
5	0x5	iconst_2	51	0x33	baload
6	0x6	iconst_3	52	0x34	caload
7	0x7	iconst_4	53	0x35	saload
8	0x8	iconst_5	54	0x36	istore
9	0x9	lconst_0	55	0x37	lstore
10	0xa	lconst_1	56	0x38	fstore
11	0xb	fconst_0	57	0x39	dstore
12	0xc	fconst_1	58	0x3a	astore
13	0xd	fconst_2	59	0x3b	istore_0
14	0xe	dconst_0	60	0x3c	istore_1
15	0xf	dconst_1	61	0x3d	istore_2
16	0x10	bipush	62	0x3e	istore_3
17	0x11	sipush	63	0x3f	lstore_0
18	0x12	ldc	64	0x40	lstore_1
19	0x13	ldc_w	65	0x41	lstore_2
20	0x14	ldc2_w	66	0x42	lstore_3
21	0x15	iload	67	0x43	fstore_0
22	0x16	lload	68	0x44	fstore_1
23	0x17	fload	69	0x45	fstore_2
24	0x18	dload	70	0x46	fstore_3
25	0x19	aload	71	0x47	dstore_0
26	0x1a	iload_0	72	0x48	dstore_1
27	0x1b	iload_1	73	0x49	dstore_2
28	0x1c	iload_2	74	0x4a	dstore_3
29	0x1d	iload_3	75	0x4b	astore_0
30	0x1e	lload_0	76	0x4c	astore_1
31	0x1f	lload_1	77	0x4d	astore_2
32	0x20	lload_2	78	0x4e	astore_3
33	0x21	lload_3	79	0x4f	iastore
34	0x22	fload_0	80	0x50	lastore
35	0x23	fload_1	81	0x51	fastore
36	0x24	fload_2	82	0x52	dastore
37	0x25	fload_3	83	0x53	aastore
38	0x26	dload_0	84	0x54	bastore
39	0x27	dload_1	85	0x55	castore
40	0x28	dload_2	86	0x56	sastore
41	0x29	dload_3	87	0x57	pop
42	0x2a	aload_0	88	0x58	pop2
43	0x2b	aload_1	89	0x59	dup
44	0x2c	aload_2	90	0x5a	dup_x1
45	0x2d	aload_3	91	0x5b	dup_x2

92	0x5c	dup2
93	0x5d	dup2_x1
94	0x5e	dup2_x2
95	0x5f	swap
96	0x60	iadd
97	0x61	ladd
98	0x62	fadd
99	0x63	dadd
100	0x64	isub
101	0x65	lsub
102	0x66	fsub
103	0x67	dsub
104	0x68	imul
105	0x69	lmul
106	0x6a	fmul
107	0x6b	dmul
108	0x6c	idiv
109	0x6d	ldiv
110	0x6e	fdiv
111	0x6f	ddiv
112	0x70	irem
113	0x71	lrem
114	0x72	frem
115	0x73	drem
116	0x74	ineg
117	0x75	lneg
118	0x76	fneg
119	0x77	dneg
120	0x78	ishl
121	0x79	lshl
122	0x7a	ishr
123	0x7b	lshr
124	0x7c	iushr
125	0x7d	lushr
126	0x7e	iand
127	0x7f	land
128	0x80	ior
129	0x81	lor
130	0x82	ixor
131	0x83	lxor
132	0x84	iinc
133	0x85	i2l
134	0x86	i2f
135	0x87	i2d
136	0x88	l2i
137	0x89	l2f
138	0x8a	l2d
139	0x8b	f2i
140	0x8c	f2l
141	0x8d	f2d
142	0x8e	d2i
143	0x8f	d2l
144	0x90	d2f
145	0x91	i2b
146	0x92	i2c
147	0x93	i2s
148	0x94	lcmp
149	0x95	fcmpl
150	0x96	fcmpg
151	0x97	dcmpl
152	0x98	dcmpg
153	0x99	ifeq
154	0x9a	ifne
155	0x9b	iflt
156	0x9c	ifge
157	0x9d	ifgt
158	0x9e	ifle
159	0x9f	if_icmpeq
160	0xa0	if_icmpne
161	0xa1	if_icmplt
162	0xa2	if_icmpge
163	0xa3	if_icmpgt
164	0xa4	if_icmple
165	0xa5	if_acmpeq
166	0xa6	if_acmpne
167	0xa7	goto
168	0xa8	jsr
169	0xa9	ret
170	0xaa	tableswitch
171	0xab	lookupswitch
172	0xac	ireturn
173	0xad	lreturn
174	0xae	freturn
175	0xaf	dreturn
176	0xb0	areturn
177	0xb1	return
178	0xb2	getstatic
179	0xb3	putstatic
180	0xb4	getfield
181	0xb5	putfield
182	0xb6	invokevirtual
183	0xb7	invokespecial
184	0xb8	invokestatic
185	0xb9	invokeinterface
186	0xba	*xxxunusedxxx*
187	0xbb	new
188	0xbc	newarray
189	0xbd	anewarray
190	0xbe	arraylength
191	0xbf	athrow
192	0xc0	checkcast
193	0xc1	instanceof
194	0xc2	monitorenter
195	0xc3	monitorexit
196	0xc4	wide
197	0xc5	multianewarray
198	0xc6	ifnull
199	0xc7	ifnonnull
200	0xc8	goto_w
201	0xc9	jsr_w

C.2 保留的操作代码

JVM标准还保留了以下的操作代码：

202 0xca `breakpoint`
254 0xfe `impdep1`
255 0xff `impdep2`

操作代码202（`breakpoint`）用于调试，而操作代码254和255则保留用于JVM本身的内部用途。它们永远不应该出现在一个存储的类文件中。事实上，带有这些操作代码的类文件应该不能通过验证。

C.3 “快速”的伪操作代码

在1995年，Sun微系统公司的研究人员提出采用内部“快速”操作代码作为提高Java编译器速度和效率的方法。通常，当访问常量池中的一个条目时，该条目必须被解析以确认其可得到和类型兼容。如果一条语句必须执行若干次，这个解析过程就会减慢计算机的速度。作为其即时（Just-In-Time，JIT）编译器，Sun提出了一组操作代码，这些操作代码假设条目已经被解析。当一个正常的操作代码被成功地执行时，其在内部可用一个“快速”伪操作代码替换，以跳过解析步骤，加快后续工作的执行速度。

这些伪操作代码永远不应出现在一个长期存储的类文件中。相反，JVM本身可将操作代码重写在一个正在执行的类中。如果做得正确，这个变化对于Java程序员甚至编译器编写者来说就完全是不可见的。所提出的优化伪操作代码包括：

203 0xcb `ldc_quick`
205 0xcd `ldc_w_quick`
206 0xce `getfield_quick`
207 0xcf `putfield_quick`
208 0xd0 `getfield2_quick`
209 0xd1 `putfield2_quick`
210 0xd2 `getstatic_quick`
211 0xd3 `putstatic_quick`
212 0xd4 `getstatic2_quick`
213 0xd5 `putstatic2_quick`
214 0xd6 `invokevirtual_quick`
215 0xd7 `invokenonvirtual_quick`
216 0xd8 `invokesuper_quick`
217 0xd9 `invokestatic_quick`
218 0xda `invokeinterface_quick`
219 0xdb `invokevirtualobject_quick`
221 0xdd `new_quick`
222 0xde `anewarray_quick`
223 0xdf `multianewarray_quick`
224 0xe0 `checkcast_quick`
225 0xe1 `instanceof_quick`
226 0xe2 `invokevirtual_quick_w`
227 0xe3 `getfield_quick_w`
228 0xe4 `putfield_quick_w`

当然，不同的实现者也完全可采用不同的优化方法或一组不同的快速操作代码。

C.4 未使用的操作代码

操作代码186由于历史原因未被使用，其先前的用法在目前的JVM版本中已不再有效。操作代码204、220及229-253在目前的JVM规范中未予分配，但在较后的版本中可能得到分配和使用。



附录D　类文件格式

D.1　概述和基本原理

在第10章我们曾简要地提到过，JVM的类文件被存储成一组嵌套的表。这实际上有点用词不当，因为在类被存储或传送时都使用相同的格式，所以在网络上接收到的类都是相同的格式，成为“.file.”。每个“file”包含了类所需要的信息，包括所有方法的字节码、类内部的域和数据，以及与类系统的其余部分进行交互作用的属性，包括名字和继承的细节。

在类文件中的所有数据都存储成8位字节或者16位、32位、64位这样多个字节的组。这在标准文档中分别以u1、u2、u4、u8来引用，但是，如果将它们仅仅看作是字节、短整数、整数及长整数或许会更容易一些。要防止采用不同存储惯例的机器之间出现混乱（比如8088及其字节交换），字节被定义为按“网络顺序”（也称为“大端字节序”或者“最高有效字节（MSB）顺序”）到来，就是最高有效字节首先到来。例如，任何JVM类的前4个字节必须是所谓的魔幻数0xCAFEBABE（显然是一个整数）。这个数被存储成4字节序列的顺序是：0xCA、0xFE、0xBA、0xBE。

类文件的顶级表包含一些（固定大小的）内务管理信息，以及5个大小变化的低级表。在不同组件之间没有填补或对齐，这使得从类文件中取出特定部分有些困难。

类文件的详细顶级格式（表D-1）如下所示：

表D-1　类文件格式的详细说明

大小	标志符	注释
整数	魔幻数	定义的值是0xCAFEBABE
短整数	次版本	定义与哪个次版本的JVM类文件相容
短整数	主版本	定义与哪个主版本的JVM类文件相容
短整数	常量池数量	在接下来的表中的最大条目数
变量	常量池	类使用的常量池
短整数	访问标志	有效访问类型（public、static、interface、等等）
短整数	该类	该类的类型的标志符
短整数	超类	该类的超类类型的标志符
短整数	接口数量	在接下来的表中的条目数
变量	接口	该类实现的接口
短整数	域数量	在接下来的表中的条目数
变量	域	作为该类组成部分而声明的域
短整数	方法数量	在接下来的表中的条目数
变量	方法	作为该类组成部分而定义的方法
短整数	属性数量	在接下来的表中的条目数
变量	属性	该类的其他属性

“魔幻数”已经描述过了，其作用是使在系统中快速辨识类文件更容易，此外并无实际意图。主版本和次版本号有助于跟踪兼容性。例如，一个非常老的类文件（或者用非常老的

Java版本编译的类文件）可能使用已不存在或语义已改变的操作代码。目前的Java版本（就是2006年版）使用了次版本号0和主版本号46（0x2e），对应于新的Java 5.0版。

这个类及其超类的域引用常量池中的条目（细节请见下一部分），并定义了当前类的名字以及直接的超类。最后，访问标志域定义了当前类的与访问相关的性质，例如，如果这个类被定义成abstract，这就阻止了其他类将其用作为new指令的参数。这些性质作为独立的标志存储在一个双字的位向量中，如下所示：

含义	位值	解　释
public	0x0001	类可被访问
final	0x0010	类不能有子类
super	0x0020	新的调用语义
interface	0x-200	文件实际上是接口
abstract	0x0400	类不能被实例化

这样，如果访问标志域是0x0601，就定义了“public”、“abstract”、“interface”。

D.2 子表结构

D.2.1 常量池

常量池被构造成为一序列的独立条目，表示程序所使用的常量。例如，一个表示整数值1010的常量将被存储在5个连续的字节中。最后4个字节就是1010的二进制表示（作为整数），而第一个字节是一个标志值，将这个条目定义为一个整数（因此要5个字节）。根据标志类型的不同，条目的大小和内部格式如下表变化：

类型	值
UTF8串	1
整数	3
浮点数	4
长整数	5
双精度数	6
类	7
串	8
域引用	9
方法引用	10
接口方法引用	11
名字和类型	12

整数、长整数、浮点数以及双精度数的结构是不言自明的。例如，一个常量池的整数入口包含5个字节，初始字节为3（将该条目定义为整数），4个字节就是整数值本身。UTF8串被存储成一个无符号的长度值（2个字节长度，允许65536个字符的串）和一个包含了串中字符值的变长度的字节数组。在Java类文件中的所有文字串，包括串常量、类名字、方法和域的名字等等，都在内部存储成UTF8串常量。

其他类型的内部域引用了常量池中其他条目的索引。例如，一个域引用包含5个字节。第一个字节是定义了一个域的标志值（值为9）。第二个和第三个字节保存了常量池中另一个条目的索引，定义了该域属于哪个类。第四个和第五个字节保存了一个“名字和类型”条目的索引，定义了名字和域。这个名字和类型条目有一个适当的标志（值为12），然后就是定义了

名字/类型的两个UTF8串的索引。

关于常量池有两个重要的警告。由于历史原因，常量池条目号0从未被使用过，初始元素的索引是1。由于这个原因，与Java中多数其他数组类型元素（以及其他类文件条目）不同，如果有k个常量池条目，则最高条目是k而不是k-1。还因为历史原因，类型为长整数或双精度数的常量池条目被当作两个条目看待。如果常量池条目6是一个长整数，则下一个池条目的索引就应该是8。在这种情况下，索引7将是无用和非法的。

D.2.2 域表

类的每个域在内部定义为一个具有如下域的表条目：

大小	标志符	注释
短整数	访问标志	该域的访问性质
短整数	名字	在常量池中的名字索引
短整数	描述符	串类型的索引
短整数	属性数	域属性的数量
变量	属性	域属性数组

名字和类型域只不过分别是常量池中名字和类型描述符串的索引。访问标志域是标志的位向量，像以前一样（下面的表给出了解释），为域定义了有效访问属性。最后，属性表定义了域的属性，就像类属性表定义了类作为整体的属性。

含义	位值	解释
Public	0x001	域可被访问
Private	0x0002	域只能被定义类所访问
Protected	0x0004	域可被类和子类所访问
Static	0x0008	类域，非实例域
Final	0x0010	域不能被改变
volatile	0x0040	域不能被缓存
transient	0x0080	域不能被对象管理器写/读

D.2.3 方法表

类中定义的每个方法都在内部描述为一个表条目，其格式几乎与前面描述的域的条目相同。唯一的差异就是对于的不同访问标志用了特定的值来表示，如下表所示：

含义	位值	解释
public	0x0001	域可被访问
private	0x0002	域只能被定义类所访问
protected	0x0004	域可被类和子类所访问
Static	0x0008	类域，非实例域
Final	0x0010	域不能被改变
synchronized	0x0020	调用是时钟同步的
native	0x0100	用本地硬件语言实现
abstract	0x0400	没有已定义的实现
Strick	0x0800	严格的浮点语义

D.2.4 属性

类文件的几乎所有部分，包括顶级表本身，都包含一个可能的属性（attribute）子表。这

个子表包含了由编译器所生成的"属性"，用以描述或支持计算。每个编译器都允许定义特定的属性，而JVM的实现还要能忽略其不认识的属性，所以这个附录无法提供一个可能属性的完全列表。另一方面，某些属性必须存在，如果JVM所要求的某个属性不存在，则JVM就不能正确运行。

每个属性被存储为以下格式的表：

大小	标志符	注 释
短整数	属性名	属性的名字
整数	属性长度	以字节数表示的属性长度
变量	信息	属性的内容

可能最显然（并且最重要）的属性就是Code属性，它作为方法的一个属性，包含了特定方法的字节码。Exceptions属性定义了特定方法可能给出的异常类型。为支持调试器，Sourcefile属性存储了创建该类文件的源文件名，LineNumberTable存储了在字节码中的哪些字节与源代码中的哪些行相对应。类似地，LocalVariableTable属性定义了（在源文件中的）哪个变量与JVM中的哪个局部变量相对应。用-g标志（在*NIX系统上）进行编译（或汇编），就通常会导致这些属性被放置到类文件。如果没有这个-g，这些属性就经常被忽略以节省空间和时间。



附录E　ASCII表

E.1 表

Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char
00	nul	01	soh	02	stx	03	etx	04	eot	05	enq	06	ack	07	bel
08	bs	09	ht	0a	lf	0b	vt	0c	ff	0d	cr	0e	so	0f	si
10	dle	11	dc1	12	dc2	13	dc3	14	dc4	15	nak	16	syn	17	etb
18	can	19	em	1a	sub	1b	esc	1c	fs	1d	gs	1e	rs	1f	us
20		21	!	22	"	23	#	24	£	25	%	26	&	27	'
28	(	29	)	2a	*	2b	+	2c	,	2d	-	2e	.	2f	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7
38	8	39	9	3a	:	3b	;	3c	<	3d	=	3e	>	3f	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G
48	H	49	I	4a	J	4b	K	4c	L	4d	M	4e	N	4f	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W
58	X	59	Y	5a	Z	5b	[	5c		5d	]	5e	^	5f	_
60	`	61	a	62	b	63	c	64	d	65	e	66	f	67	g
68	h	69	i	6a	j	6b	k	6c	l	6d	m	6e	n	6f	o
70	p	71	q	72	r	73	s	74	t	75	u	76	v	77	w
78	x	79	y	7a	z	7b	{	7c	|	7d	}	7e	~	7f	del

注意：ASCII0x2.0（十进制32）是一个空格（“”）字符。

E.2 历史和概述

ASCII，即美国标准信息交换码（American Standard Code for Information Interchange）几十年来一直是以二进制格式对字符数据进行编码的最常用标准（它在1963年形成，在1968年国际化）。其他曾经常用的格式如EBCDIC和Bauot在很大程度上已经成为历史。初始的ASCII字符集定义了一个7位的标准，用于对128个不同的字符进行编码，包括（美国英语中）全部大写和小写字母的集合，以及很多符号和标点符号。此外，在ASCII表中前32个项定义了大多数的不可打印的控制字符（control characters），如退格（backspace，0x08）、[水平]制表符（horizontal tab，0x09）、[垂直]制表符（vertical tab，0x10），甚至一个有声的“铃声(bell)”（现在通常是一次响铃beep，0x07）。遗憾的是，ASCII对非英语的语言支持得不好，甚至对很多常见的有用符号如→、≤以及英镑符号(£)也不支持。

由于几乎所有的计算机都是存储8位字节，所以字节0x80...0xFF这些没有被解释成标准字符的字节值就常常用于对ASCII表进行机器无关的专有扩展。例如，字母Ö用于德文中而不用于英文中。微软已经定义了一个扩展的ASCII表（此外还有由多种基于Windows程序所使用的若干个不同集合），该表用项0x94来表示这个值，作为一个相当完整的德文专用字符集的组成部分。这个表还将项0xE2定义成大写的Γ，但奇怪的是却没有为小写的γ定义一个项。我猜测，对于字符集的设计者来说，说德语的市场要比说希腊语的市场更重要。对比起来，苹果对于扩展的字符没有定义意思（至少对于其在OS X环境下的“终端”环境）。

问题的核心是：只有256种不同存储模式的单个字节不能存储足够多的不同字符。Java的解决方案是采用一个更大的字符集（Unicode）并将它们存储成2字节的量。

词 汇 表

80x86 family（80x86系列） Intel制造的一个系列芯片，从Intel 4004开始，并历经8008、8088、80086、80286、80386、80486，以及各种奔腾芯片。这些芯片构成了IBM-PC的基础，其后继就是当今最常用的芯片体系结构。

absolute address（绝对地址） 在基于8086的计算机中，通过对段:偏移存储地址进行组合而获得的20位地址。

abstract （抽象） 一个不包含直接的实例而只有子类的类。或者是一个必须在子类中实现的未实现的方法。

accumulator（累加器） 一个为进行高速算术运算，特别是加和乘而指定的单个寄存器。在80x86计算机中就是[E]AX寄存器。

actual parameter（实际参数） 在对函数或方法进行调用时，用于替代形式参数的实际值。

address （地址） 在存储器中的一个位置，或者说是用于指定存储器中某位置的数。

addressing mode （寻址模式） 解释位模式的方式，用来为语句定义实际的操作数。例如，位模式0x0001可指定实际常数1、第1个寄存器、存储器位置1的内容，等等。参见各个模式：immediate mode（立即模式）、register mode（寄存器模式）、direct mode（直接模式）、indirect mode（间接模式）、index mode（变址模式）。

algorithm （算法） 一个按步进行的、意义明确的过程。用于达到期望的目标或执行一个计算。

ALU 典型计算机体系结构中的一个部件，在其中要执行算术和逻辑操作。是CPU的一部分。

American Standard Code for Information Interchange （美国信息交换标准码） 参见ASCII。

AND（与（AND）） 一个布尔函数，当且仅当所有参数都是真时返回真，否则就返回假。一个与门是一个硬件电路，它对输入电信号实现了与函数。

applet（应用程序（applet）） 一个小的、可移植的程序，典型地是作为Web页的一部分。

Arithmetic and Logical Unit（算术和逻辑单元） 见ALU。

arithmetic shift（算术移位） 一个移位操作，其中最左端（最右端）位被复制以填充空出的位置。

array（数组） 一个派生的类型。是由一个整数索引的一组相同类型的子元素。

ASCII（ASCII） 用二进制格式表示字符数据的一种标准方式。ASCII集为字母、数字以及一些常用标点符号定义了7位的模式。

assembler（汇编器） 一个将用汇编语言写成的原文件转换成可执行文件的程序。

assembly language（汇编语言） 一种低级语言，其中每条人可读的语句精确地对应一条机器指令。不同的计算机类型有不同的汇编语言。

attributes（属性） JVM类文件格式的一种数据结构，用于存储多方面的信息。

backward compatibility（向后兼容） 计算机或者系统复制以前机型的操作的能力。例如，奔腾对于8088是向后兼容的，所以对于为最初IBM-PC写的程序它也能运行。

base（基） 1.（基数）用数制中数字的位置表示的数值。例如，二进制的基数是2，十进制的基数是10，而十六进制的基数是16。2.（基极）在晶体管中的一个电连接，用于控制从发射极到集电极的电流。

BAT（BAT） 将存储管理器内部的逻辑地址转换到固定物理存储块的方法。

BCD（BCD） 在80x86系列的数字处理器中所采用的一种方法。在这种方法中，一个数的每个十进制数字被写成相应的4位二进制模式。例如，数4096写成BCD数就是0100 0000 1001 0110。

big-endian（大端字节序） 一种存储格式，其中最高有效位存储成字的第一个和最高序的位。依照大端字节序格式，数32770将被存储成二进制10000010。

binary（Binary（二进制/二元）） 1. 一种计数制，其中所有数字都是1或者0，并且连续数字的权值是2倍的关系。数31写成二进制数就是11111。2. 只接受两个操作数的数学操作符，如加法。

Binary Coded Decimal（二进制编码的十进制数） 见BCD。

bit（位） 一个二进制数字，是计算机内信息的基本单位。一个位可有两种取值：0或者1。

bitwise（按位） 一种布尔操作。在像字节和整数的多位结构中，这种操作是分别对每个位独立地进行的。

block address translation（模块地址转换） 见BAT。

boolean logic（布尔逻辑） 依乔治·布尔命名的一种逻辑系统，其中的操作定义成二进制量上的函数并按此执行。

branch（转移） 一种机器指令，这种指令改变程序计数器（PC）的值因而导致计算机在一个不同的存储位置开始执行。一个等价的术语是goto。

branch prediction（转移（分支）预测） 一种优化技术，这种技术通过预测条件转移指令是否发生转移来加速计算机的运行速度。

bus（总线） 典型机器体系结构的一个组件，承担CPU、存储器以及外设之间的连接任务。

busy-waiting（总线等待） 通过检测事件是否在循环内部已经发生来等待期望的事件。与中断比较。

byte（字节） 8个位的集合。用作为一个存储单位，以表示存储器大小或者寄存器容量。

bytecode（字节码） JVM的机器语言。

cache memory（高速缓冲存储器） 一块高速存储器，用于存储经常访问的项以提高存储器的总体性能。

Central Processing Unit（中央处理单元） 见CPU。

CF 进位标志，当最近的操作生成一个向寄存器外的进位时（比如两个数相加得到的和过大时），就将其置位。

CISC 一种计算机设计观点，即使用很多复杂的专用指令。与RISC相对。

class（类） 一组域和方法，定义了面向对象编程环境中的对象类型。

class files（类文件） JVM采用的一种格式，用于将类（包括记录和接口）存储到长期存储器或通过网络发送。

class method（类方法） 由面向对象系统所定义的方法，是作为类的特性而不是该类的任何实例的特性。

class variable（类变量） 由面向对象系统所定义的变量，是作为类的特性而不是该类的任何实例的特性。

clock signal（时钟信号） 计算机用来对事件进行同步和度量时间推移的电信号。

CLR 微软的.NET框架的虚拟机。

code（代码） 可执行的机器指令，与数据相对。

collector（集电极） 标准晶体管的一部分，除非基极电流关闭，将会有从发射极到集电极的电流。

comment（注释） 编程语言中的语句，虽然被计算机所忽略，但却可能包含了可读的有用信息。在jasmin中，注释开始于分号（;），直至行尾。

Common Language Runtime（通用语言运行时） 见CLR。

compiler（编译器） 一种程序，用于将用高级语言写的源文件转换成可执行文件。

complete/writeback（完成/写回） 流水线体系结构中的一个典型阶段，在这个阶段将操作的结果存储于目标位置。

Complex Instruction Set Computing（复杂指令集计算） 见CISC。

conditional branch（条件转移） 一种转移（如ifle），根据当前机器状态决定是否转移。

constant pool（常量池） 特定JVM类使用的一组常量，存储在类文件中。

control characters（控制字符） 值低于0x20的ASCII字符，表示非打印字符，如回车或响铃。

Control Unit（控制单元） CPU的组成部分，其功能是将数据移入和移出CPU和确定执行哪条指令。

CPU 计算机的心脏，在其中发生计算，程序被执行。通常由控制单元和ALU组成。

data memory（数据存储器） 用于存储程序数据（如变量）而不是程序代码的存储器。

decimal（十进制） 基数为10。写数字的通常方式。

derived type（派生类型） 通过将基本类型相结合而建立起来的数据表示类型。例如，一个分数类型可由分子和分母两个整数派生而成。

destination（目标） 数据的去处。例如，在指令istore_3中，目标是局部变量#3。

destination index（目标变址） 在80x86系列中的一个寄存器，用于控制串原始操作的目的地。

destination operand（目标操作数） 定义了指令目标的操作数。在奔腾指令集中，如在MOV AX,BX中，目标操作数通常为第一个操作数。

device（设备） 外设的另一个名字。

device driver（设备驱动器） 一个程序（或者操作系统的组成部分，用于控制设备）。

diode（二极管） 一种电气部件，只能允许电从一个方向通过。是晶体管结构的组成部分。

Direct Memory Access（直接存储器访问） 计算机拥有的一种能力，即让数据在主存储器与外设（如图形卡）之间移动而无需通过CPU。

direct mode（直接模式） 一种寻址模式，其中的位模式被解释为保存了所需操作数的存储位置。在直接模式中，位模式0x0001被解释为存储位置1。

directive（汇编指令） 汇编语言中的语句，在jasmin中，以句号（.）开头，这种语句不转换成机器指令但给汇编器指示。.limit就是汇编指令的例子。

dirspatch（分派） 流水线体系结构的一个典型状态，此阶段要做的是：计算机分析指令以确定是什么类型的指令，从适当的位置获得源参数，并为实际执行准备指令。

dopants（掺杂物） 有意加入到半导体（如硅）中的杂质，以影响其电特性。引入这些杂质就能构造出二极管和晶体管，这些都是像计算机这种电子设备的关键部件。

DRAM（DRAM） 一种RAM，需要不断地刷新以保持数据。比SRAM慢但便宜。与所有的RAM一样，如果电源关闭，则存储器丢失其数据。

dst 目标（destination）的缩写。

Dynamic RAM（动态RAM） 见DRAM。

EEPROM 一种混合存储器，结合了RAM的现场可编程性和ROM的数据持久性。

Electronically Erasable Programmable ROM（电可擦除可编程ROM） 见EEPROM。

embedded system（嵌入式系统） 一种计算机系统，其中的计算机是更大环境的组成部分而不是独立可使用的工具。一个例子是运行DVD播放器的计算机。

emitter（发射极） 标准晶体管的组成部分，除非基极电流关闭，将会有从发射极到集电极的电流。

EPROM 一种PROM，一般可通过暴露于高能紫外光中几秒钟来擦除其内容。这些存储器是可重编程的，但不是现场可重编程。

Erasable Programmable ROM（可擦除可编程ROM） 见EPROM。

execute（执行） 1. 运行一个程序或机器指令。2. 在流水线体系结构中的一个典型状态，在此阶段计算机运行一条以前取出（并分派）的指令。

exponent（指数） 在IEEE浮点表示中的一个域，用于控制2的幂，该幂要乘以尾数。

extended AX register（扩展AX寄存器） 在80386或80x86系列较后来的芯片（包括奔腾）上的32位累加器。

fetch（取指） 1. 装入一条机器指令以准备执行。2. 流水线体系结构的一个典型状态，在此阶段计算机从主存储器装入一条指令。

fetch-execure cycle（取指－执行周期） 是指这样的过程：计算机通过取指得到要执行的指令，执行该指令，然后再取序列中的下一条指令，直到程序结束。

Fibonacci sequence（斐波那契序列） 序列1,1,2,3,...，其中每一项是前面紧邻两项之和。

fields（域） 在记录或类中命名的数据存储位置。

flags（标志） 用于存储数据的二进制变量。参见flag register（标志寄存器）。

flags register（标志寄存器） 在CPU中的一个特殊寄存器，保存了一组与当前计算状态相关的二进制标志。例如，如果在算术计算中机器发生溢出，一个标志（一般称为“溢出标志”即OF）将被置为1。一个后来的条件转移指令可检查这个标志。

Flash（闪存存储器） 一种混合存储器，将RAM的现场可编程性与ROM的数据持久性相结合。通常用于笔型U盘和数字相机中。

floating point（浮点） 1. 计算机中存储的任何非整数值。2. 一种特定的格式，用于存储二进制科学表示法的非整数值。值被存储成尾数与2的指数幂的乘积。

Foating Point Unit（浮点单元） 见FPU。

FPU（FPU） 配属于CPU的专用硬件，用以处理浮点（不是整数）计算。现在有点罕见了，因为多数CPU能在自身处理浮点操作。

foraml parameter（形式参数） 用在函数或方法定义中的变量，用于作为以后实际参数的占位符。

garbage collection（垃圾收集） 回收不再使用的存储位置。在JVM中是自动进行的。

gates（门） 实现了布尔函数的电路。

goto 参见branch（转移）。

Harvard architecture（哈佛体系结构） 一种非冯·诺依曼体系结构，其中的代码存储（用于程序）与数据存储（用于变量）是分开的。

hexadecimal（十六进制） 基为16。一种数制，其中所有数字都是0-9或字母A-F，并且连续的数字之间是16倍的关系。十六进制数33可写成0x31。

high（高） 1. 寄存器或数据值的高位部分。特别是，在8088上的通用寄存器的最高有效字节。2. 见high-level(高级)。

high-level（高级） 指Java、C++、或Pascal这样的高级语言，其中的一条语句可能对应于若干条机器语言指令。

hybrid memory（混合存储器） 一种存储器，其设计将RAM的现场可重写性和ROM的数据持久性结合起来。例如，见EEPROM或Flash。

immediate mode（立即模式） 一种寻址模式，其中的位模式被解释为常数操作数。在立即模式中，位模式0x001就是常数1。

implement（实现） 接口的实现就是遵循接口而不是接口的一个实例。

index mode（变址模式） 一种寻址模式，其中一个位模式被解释为针对一个存储地址的偏移，该存储地址存在一个寄存器中。

indirect address register（间接寻址寄存器） 一个用于间接或变址模式的寄存器，如奔腾的BX或Atmel的Y寄存器。

indirect mode（间接模式） 一种寻址模式，其中的一个位模式被解释为保存了指向实际操作数指针的存储位置。在间接模式中，位模式0x0001将被解释为存储在存储位置1的值。

infix（中缀） 一种写表达式的方法，其中的二元操作符位于其参数之间，如3+4。请比较后缀和前缀。

initialize（初始化） 在使用前设置一个特定的初始值，或者调用一个函数来执行这个任务。

instance method（实例方法） 一个方法，由面向对象系统作为一个性质定义，而不是由其控制类所定义。

instance variable（实例变量） 一个变量，由面向对象系统作为一个性质定义，而不是由其控制类所定义。

instruction（指令） 一般来说是对计算机的一个命令。在机器代码中，就是表示单个操作的位模式。在汇编语言中，就是一种能直接转换成机器指令的语句。

instruction pointer（指令指针） 见IP。

instruction queue（指令队列） 一组有序的指令，等待被装入，或者已经被装入并等待被执行。

instruction register（指令寄存器（IR）） CPU内的寄存器，保存当前指令以待分派和执行。

instruction set（指令集） 某特定CPU能执行的一组操作。

integrated circuit（集成电路） 一个制造在单个硅芯片上的电路而不是很多分立元件。

interface（接口） 一个抽象的类，定义了不同对象共享的行为，但在正常的继承结构之外。

interrupt（中断） 1. 一小段预先建立的代码，当特定事件发生时执行。2. 对CPU的通知，告诉CPU事件已经发生，这一段代码应该执行了。

interrupt handler（中断处理程序） 一个用于处理期望事件而无需忙碌-等待开销的系统。

invoke（调用） 执行一个方法。

I/O controller（I/O控制器） 典型机器体系结构的一个部件，用于控制一个特定的外设进行输入和输出。I/O控制器通常接受和解释来自总线的信号，并照顾在操作特定部件如硬驱时的细节。

I/O registers（I/O寄存器） 用于发送信号到一个I/O控制器的寄存器，尤其是在Atmel AVR上。

IP 即指令指针，是一个标准的CPU寄存器，保存了正在执行的当前指令的位置。

jasmin（jasmin） 由Meyer和Downing为JVM写的汇编器，本书的主要教学语言。

Java Virtual Machine（Java虚拟机） 见JVM。

JIT compilation（JIT编译） 一种加速JVM程序执行的技术，是通过将每条语句转换成等价的本地机器语言指令实现的。

Just In Time（即） 见JIT编译。

JVM 一个作为Java编程语言基础的虚拟机，本书的主要教学机器。

label（标号） 在汇编语言中，是针对特定代码行的可读标记，使得该行可作为转移指令的目标。

latency（等待时间） 完成某事所需要的时间。在一个指令等待时间为1μs的计算机上，执行一条指令至少需要这些时间。

linear congruential generator（线性同余数生成器） 一个通用的伪随机数生成器，生成器连续返回的是形为newvalue=(a · oldvalue+c)%m的等式的值。

link（链接） 将存储于磁盘上的一组字节码（或机器指令）转换成可执行格式的过程。

little-endian（小端字节序） 一种存储格式，其中最低有效位被存储在一个字的第一个和最高序的位。在小端字节序格式中，数32770被存储为二进制数01000001。

llasm（llasm） 与微软的.NET框架一起使用的汇编器。

load（装入） 将一组字节码（或机器指令）从磁盘转移到存储器的过程。

logical address（逻辑地址） 存储在寄存器中的位模式，用于在由任何存储管理或虚拟存储例程解释之前访问存储器。

logical memory（逻辑存储器） 由一组逻辑地址定义的地址空间，与由存储管理器存储数据的物理存储器相区别。

logical shift（逻辑移位） 一种移位操作，其中新空出的位置用0值来填充。与算术移位相比较。

long（长整数） 在Java或jasmin中，一种64位（两个字）整数类型的数据存储格式。

low-level（低级） 一种像jasmin或其他汇编语言的语言，其中单个语句对应于单个机器语言指令。

machine code（机器码） 见机器语言。

machine cycle（机器周期） 计算机的一个基本时间单位，典型地定义为执行单条指令所需时间，或者也可以定义为系统时钟的时间单位。

machine language（机器语言） 计算机程序的基本指令的二进制编码。机器语言一般不是由人写的，而是由其他程序如编译器或汇编器生成的。

machine state register（机器状态寄存器） 一个将计算机的总体状态描述为一组标志的寄存器。

mantissa（尾数） 一个浮点数的分数部分，即将要用一个包含2的某次幂的比例因子相乘。

math coprococessor（数学协处理器） 一个辅助芯片，通常用于浮点计算，而ALU处理的

是整数计算。

memory-manager（存储管理器） 一个用于控制程序访问物理存储器的系统。它通常能提高性能、增加安全性、并增大程序能使用的存储量。

memory-maped I/O（存储器-映射的I/O） 一种执行I/O的方法，其中I/O控制器自动读特定的存储位置而不使用总线。

microcontroller（微控制器） 一种小的计算机，通常是嵌入式系统的组成部分，而不是独立可编程的计算机的组成部分。

microprogramming（微程序设计） 将计算机的（复杂）指令集实现为一序列较小的类RISC的指令。

Microsoft Intermediate Language（微软中间语言） 见MSIL。

MIMD 计算机在两个不同部分同时执行两个不同指令的能力。

MMX instructions（MMX指令） 在80x86系列芯片的较后的模型上实现SIMD并行性的指令。

mnemonic（助记符） 一种可读的汇编语言程序的一部分，对应于特定的操作或操作代码。

mode（模式） 见addressing mode（寻址模式）。

modulus（模数） “余数”的形式化数学定义。

monitor（监视器） JVM的一个子系统，用于确保在同一时间只有一个方法/线程能访问一块数据。

Monte Carlo simulation（蒙特卡洛模拟） 一种通过重复使用随机数来探索大规模解空间的技术。

most significant（最高有效） 对应于基数的最高幂的数字或字节。例如，对于数361402，数字3就是最高有效数字。还请见大端字节序、小端字节序。

motherboard（母板） 计算机板，CPU和多数至关重要的其他部件都置于其上。

MSIL 对应于JVM字节码的语言，是微软.NET框架的基础。

MSR 见机器状态寄存器。

nonvolatile memory（非易失存储器） 即使是在掉电后所存储的数据仍存在的存储器。

Non-Volatile RAM（非易失RAM） 见NVRAM。

normalized form（规格化形式） 典型浮点数的标准格式（根据IEEE 754标准）。这种格式包括：一个符号位、一个8位偏置指数、以及一个23位尾数（隐含由1打头）。

n-type（n型） 一种用能提供电子的物质掺杂的半导体，使这些电子能传导电流。

null（null） 一个指定的地址，不特别引用任何东西。

NVRAM（NVRAM） 结合了RAM的现场可编程性和ROM的持久性的混合存储器。

nybble（半字节） 4个位的一组，指的是一个十六进制数字。用作为表示存储器大小的单位，或者寄存器容量。很少使用。

object-oriented programming（面向对象的编程） 一种编程风格，由于Smalltalk、C++以及Java等语言而广为人知。在这种编程风格中，程序由相互作用的类和对象组成，通信通过调用特定对象上的方法来完成。

octal（八进制） 基为8。现在很少使用了。

OF 溢出标志，当最近的带符号数操作产生对于寄存器而言过大的解时，就将其置位。

offset（偏移） 存储器中两个位置之间的距离。通常与基寄存器（如在8088存储器段中）一起使用，或者也可作为转移指令改变PC的量。

opcode（操作代码） 与机器代码中特定操作相对应的字节。与mnemoinc（助记符）比较。

operands（操作数） 特定操作的参数。例如，ADD指令通常接受两个参数。然而，在JVM中，iadd不接受任何参数，因为两个参数都已经在堆栈上了。

operating system（操作系统） 一个控制程序，用于控制机器对用户级程序的可用性，并在适当的时间发起和恢复。操作系统的常见例子是Windows、Linux、MacOS及OS X。

operator（操作符） 指明哪个数学或逻辑函数应该被执行的符号或代码。例如，操作符+通常是指加法。

OR（或） 一种布尔函数，当且仅当有一个参数为真时返回真，否则返回假。或门是针对输入电信号实现了或函数的硬件电路。

overclocking（超频） 试图用比芯片的额定速率快的时钟运行计算机。

overflow（溢出） 宽泛地说，就是当一个算术操作产生一个过大的值而不能存储到目标时。例如，两个8位数相乘就可能得出一个16位的结果，存储到8位的目标寄存器就会溢出。

page（页） 存储管理系统中所用的存储块。

page table（页表） 存储管理器使用的表，用来确定哪个逻辑地址对应于哪个物理地址。

paging（分页） 将存储器分成“页”。或者说，是计算机在主存储器和长期存储器之间来回移动页的能力，用以扩展程序可得到的存储量并提高性能。

parallel（并联） 在一个电路中，两个部件如果每个都有一个独立的路径，使得电流能独立流过，则这两个部件就是并联的。参见串联。

parallelism（并行性） 计算机在同一时间执行多个操作的能力。参见MIMD和SIMD。

PC 一个保存了当前正在被执行指令的存储位置的寄存器。改变这个寄存器的值将会导致从一个不同的位置装入下一条指令。这也就是转移指令的工作原理。

peripheral（外设） 计算机的一个部件，用于读、写、显示或存储数据，或者更一般地与外部世界交互。

pipelining（流水线操作） 像一个装配流水线一样将一个过程（一般是机器指令执行）分裂成若干个阶段。例如，计算机可能在执行一条指令的同时取下一条指令。一个典型的流水线体系结构可同时执行若干条不同的指令。

platter（盘片） 在硬驱中的一个独立的存储表面。

polling（轮询） 进行测试以发现某事件是否已经发生。见busy-waiting（忙碌-等待）。

port-mapped I/O（端口映射的I/O） 一种执行I/O的方法，其中与I/O控制器的通信是通过配属于主总线的特定端口实现的。

postfix（后缀） 一种写表达式的方法，其中的二元操作符放在其操作数的后面，如34+。可与infix（中缀）或prefix（前缀）比较。

prefetch（预取） 在前一条指令执行完毕前就取指令。一种粗糙形式的流水线操作。

prefix（前缀） 一种写表达式的方法，其中的二元操作数放在其操作符的前面，如+34。可与中缀或后缀比较。

primordial class loader（原始类加载器） 主要的类加载器，负责加载和链接JVM中所有的类。

program counter（程序计数器） 见PC。

programmable ROM（可编程ROM） 现场可编程（但不是可擦除的）ROM。与传统ROM芯片不同，这些芯片可进行少量的编程而无需建立全部的生产线。

programming models（可编程模型（模式）） 一种对计算机体系结构和能力所定义的概观，由于安全方面的原因故有限制。

PROM（PROM） 见Programmable ROM。
protected mode（保护模式） 一种编程模式，其中程序的能力受存储管理和安全问题所限制。对在多处理系统上的用户级程序有用。
pseudorandom（伪随机） 一个由算法生成的近似随机性。
p-type（P型） 一种用能捕获电子（即“提供空穴”）的物质掺杂的半导体，使这些电子不能传导电流。
radix point（基数小数点） “十进制小数点”思想的推广，是针对不限于10的基数。
RAM 能够在任意位置读出和写入的存储器。RAM是易失性的，即掉电后则存储器中数据也会丢失。
Random Access Memory（随机存取存储器） 见RAM。
Read-Only Memory（只读存储器） 见ROM。
real mode（实模式） 一种编程模型，其中程序能使用机器的全部能力，绕开了安全性和存储管理。主要用于操作系统和其他的管理级程序。
record（记录） 一组命名的域，但没有方法。
reduced instruction set computing（精简指令集计算） 见RISC。
register（寄存器） CPU内部的一个存储位置，用于存储目前正在操作的数据和指令。
register mode（寄存器模式） 一种寻址模式，其中的一个位模式被解释为一个特定的寄存器。在寄存器模式中，位模式0x0001就会被解释成第一个寄存器。
return（return） 在子例程结束时将控制传递回调用环境的过程（也常指返回所用的指令）。
RISC 一种计算机设计观点，即使用一些短的通用指令。与CISC相对。
ROM 只能读出不能写入的存储器。ROM是非易失性的，即若掉电则存储器中的数据仍能保持。
roundoff error（舍入误差） 当一个浮点表示不能表示精确的值时所发生的误差，通常是因为定义的尾数过短。期望的值将被“舍入”到最接近的可表示的量。
SAM 不能以任意顺序访问，却必须以预先定义的顺序访问的存储器，如磁带记录。
seed（种子） 用于开始生成伪随机数序列的值。
segment（段） 宽泛地说，是存储器中的一个连续区域。更明确地说，是由80x86系列的一个段寄存器所引用的一个存储器区域。
segment:offset（段：偏移） 在英特尔8088（或以后的模型）的16位寄存器上表示20位逻辑地址的一种可选择的方式。
segment register（段寄存器） 80x86系列的一个寄存器，用于为代码、堆栈及数据定义存储块，并扩展可得到的地址空间。
semiconductor（半导体） 一种电材料，处于导体和绝缘体之间，可用于生成二极管、三极管、及集成电路。
Sequential access memory（顺序访问存储器） 见SAM。
series（串联） 在一个电路中，两个部件如果只有一条路径使得电流流过两个部件，则这两个部件就是串联的。参见parallel（并联）。
SF 符号标志，当最近操作产生一个负的结果时将其置位。
shift（移位） 一种操作，其中寄存器中的位被移动到（左边或右边的）相邻位置。
sign bit（符号位） 在有符号数的表示中，用于指明值是正数还是负数的位。通常符号位为1用于表示负数。对于整数和浮点数都是这样。

signed（有符号的） 一个可以为正或为负的量，与只能为正的无符号的量相对。

SIMD（SIMD） 计算机在同一时间在不同的数据块上执行同一条指令的能力。例如，将存储器中若干个元件同时清零。

Single Instruction Multiple Data（单指令多数据） 见SIMD。

source（源） 数据的来处。例如，在指令iload_3中，源就是局部变量#3。

source index（源变址） 在80x86类列中的一个寄存器，用于控制串原始操作的来源。

source operand （源操作数） 定义了一条指令的操作来源的操作数。在奔腾指令集中，源操作数通常就是在MOV AX, BX中的源操作数。

src 源的简写。

S-R flip-flop（S-R触发器） 用于将单个位的值作为一组自增强晶体管的电压进行存储的电路。

stack（堆栈） 一种数据结构，其中的元素只能在一端插入（压入）和删除（弹出）。堆栈在机器体系结构中的用途是作为短期存储器生成和销毁新位置的方式。

stack frame（堆栈帧） 一种基于堆栈的内部数据结构，用于存储当前正在执行程序或子例程的局部环境。

stack machine（状态机） 一个计算机程序，其中计算机仅根据接收到的特定输入或事件从一种命名的“状态”变化到另一个状态。

static（静态的） 见class method（类方法）。

static RAM（静态RAM） 见SRAM。

string primitive（串基元） 80x86系列上的一种操作，用于对像串这样的字节数组进行移动、拷贝或其他处理。

structure（结构） 见record（记录）。

subroutine（子例程） 一个被封装的代码块，在程序执行的过程中可从若干个不同的位置调用，之后控制再被传递回被调用的地点。

superscalar（超标量） 一种通过复制流水线或流水线段来增强性能，以达到MIMD并行性的方法。

supervisor（管理模式） 一种特权编程模型，在这种模式下，计算机有能力以一种方式控制系统，而这种控制方式通常是安全体系结构所禁止的。管理模式通常为操作系统所使用。参见real mode（实模式）。

this 在java中，就是其实例方法正在被调用的当前对象。在jasmin中，引用this的对象总是作为局部变量#0传递。

throughput（吞吐量） 每个时间单位能完成的操作数量。这与在多处理器上的等待时间（latency）或许不同，例如，多处理器能在相等的等待时间内同时完成若干个操作，实际上达到了期望吞吐量的若干倍。

Throwable（可抛出） 在JVM中，athrow指令抛出的一个对象、一个异常或者错误。

timer（定时器） 一种对时钟脉冲进行计数的电路，以确定流失的时间量。

time-sharing（时间共享） 一种形式的多重处理技术，其中CPU在一个时刻为一个程序运行一小段时间，这就给出了多个程序都在同时运行的幻觉。

two's complement notation（二进制补码表示） 一种存储带符号整数的技术，使得加法操作对于正数和负数都一样进行。在二进制补码表示中，−1的表示就是所有位都是1的位向量。

typed（带类型的） 就是在计算、表示、或操作中，被处理数据的类型影响到结果的合法性和正确性。例如，istore_0就是一个带类型的操作，因为它只存储整数。相反，dup操作是不带类型的，因为它复制任何的堆栈元素。

U 在英特尔奔腾处理器上的两个5段流水线之一。

unary（一元的） 一种只接受一个操作数的数学操作，如取反或求余弦。

unconditional（无条件的） 总是，比如在一个无条件转移中，总是转移。

Unicode 一个字符表示的集合，包括了比ASCII更大量的字母。见UTF-16。

unsigned（无符号的） 一个只能是正数或负数的量，与可正可负的带符号量相反。

update mode（更新模式） 一种寻址模式，其中寄存器的值在访问后被更新，如递增到下一个数组位置。

user mode（用户模式） 一组编程模式，其中程序的能力受到安全体系结构和操作系统的限制，以防止两个程序做有害的交互作用。

UTF-16 与ASCII不同的字符编码方式，其中字符被作为16位的量存储。这就可允许在字符集中容纳多达65535种不同的字符，足够将大量的非英文和非拉丁字符包括进来。

V 在英特尔奔腾上两个5段流水线之一。

verifier（验证器） 加载类文件并进行验证的阶段，或者执行这种验证的程序。

verify（验证） 在JVM上，就是确认一个方法或类可成功地运行而不会引起安全问题的过程。例如，试图将先前作为浮点数装入的值作为整数值存储就会引起一个错误，但在类文件被验证时可捕获到这个错误。

virtual address（虚拟地址） 在虚拟存储系统中的一个抽象地址，在实际存储器中可存在或不存在（并可在长期存储器中存在）。参见logical mode（逻辑地址）。

virtual address space（虚拟地址空间） 见virtual address（虚拟地址）。

virtual memory（虚拟存储器） 计算机解释逻辑地址并将其转换成物理地址的能力。也就是计算机访问实际上不在主存储器中的逻辑地址的能力，这是通过将逻辑地址空间的某些部分存储到磁盘并按需要装入到存储器来实现的。

virtual segment identifier（虚拟段标志码） 见VSID。

VSID 一个位模式，用于将逻辑地址扩展成更大的地址空间，为存储管理器所使用。

watchdog timer（看门狗定时器） 一种定时器，当触发时，就会复位计算机，或者进行检验以确认机器没有进入一个无限循环，或者无反应的状态。

word（字） 在机器中数据处理的基本单位，一般是通用寄存器的大小。在写这本书的时候（2006年），多数可买到的计算机的典型值是32位。

ZF 零标志位，当最近操作的结果为0时将其置位。

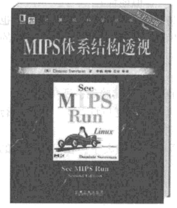
经典藏书



作者：David A.Patterson;
John L.Hennessy
译者：郑纬民　等
中文版：978-7-111-20214-1
定价：75.00
英文版：978-7-111-19339-3
定价：85.00



作者：John L. Hennessy;
David A. Patterson
英文版：978-7-111-20378-X
定价：78.00



作者：Dominic Sweetman
译者：李鹏　鲍峥　石洋
中文版：978-7-111-23362-6
定价：48.00
英文版：978-7-111-20681-9
定价：65.00



作者：Wayne Wolf
英文版：978-7-111-20416-6
定价：65.00
中文版：2009年出版
译者：汪东升



北京培生信息中心
中国北京海淀区中关村大街甲59号
人大文化大厦1006室
邮政编码：100872
电话：(8610)82504008/9596/9586
传真：(8610)82509915

Beijing Pearson Education
Information Centre
Room1006,CultureSquare No.59 Jia, Zhongguancun Street
Haidian District, Beijing, China100872
TEL: (8610)82504008/9596/9586
FAX: (8610)82509915

尊敬的老师：

您好！

为了确保您及时有效地申请教辅资源，请您务必完整填写如下教辅申请表，加盖学院的公章后传真给我们，我们将会为您开通属于您个人的唯一账号以供您下载与教材配套的教师资源。

请填写所需教辅的开课信息：

采用教材			□中文版 □英文版 □双语版
作　者		出版社	
版　次		ISBN	
课程时间	始于　年　月　日	学生人数	
	止于　年　月　日	学生年级	□专科　□本科1/2年级 □研究生　□本科3/4年级

请填写您的个人信息：

学　校			
院系/专业			
姓　名		职　称	□助教 □讲师 □副教授 □教授
通信地址/邮编			
手　机		电　话	
传　真			
official email(必填) (eg:XXX@ruc.edu.cn)		email (eg:XXX@163.com)	
是否愿意接受我们定期的新书讯息通知：	□是　□否		

系 / 院主任：________（签字）

（系 / 院办公室章）

______年______月______日

Please send this form to: Service.CN@pearson.com

Website: www.pearsonhighered.com/educator

教师服务登记表

尊敬的老师：

您好！感谢您购买我们出版的＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿教材。

机械工业出版社华章公司为了进一步加强与高校教师的联系与沟通，更好地为高校教师服务，特制此表，请您填妥后发回给我们，我们将定期向您寄送华章公司最新的图书出版信息！感谢合作！

个人资料（请用正楷完整填写）

<table>
<tr><td>教师姓名</td><td></td><td>□先生
□女士</td><td>出生年月</td><td></td><td>职务</td><td></td><td colspan="2">职称：□教授 □副教授
□讲师 □助教 □其他</td></tr>
<tr><td>学校</td><td colspan="3"></td><td>学院</td><td colspan="2"></td><td>系别</td><td></td></tr>
<tr><td rowspan="2">联系电话</td><td colspan="3" rowspan="2">办公：
宅电：
移动：</td><td>联系地址及邮编</td><td colspan="4"></td></tr>
<tr><td>E-mail</td><td colspan="4"></td></tr>
<tr><td>学历</td><td></td><td>毕业院校</td><td></td><td colspan="2">国外进修及讲学经历</td><td colspan="3"></td></tr>
<tr><td>研究领域</td><td colspan="8"></td></tr>
<tr><td colspan="3">主讲课程</td><td colspan="2">现用教材名</td><td>作者及出版社</td><td>共同授课教师</td><td colspan="2">教材满意度</td></tr>
<tr><td colspan="3">课程：
□专 □本 □研
人数： 学期：□春□秋</td><td colspan="2"></td><td></td><td></td><td colspan="2">□满意 □一般
□不满意 □希望更换</td></tr>
<tr><td colspan="3">课程：
□专 □本 □研
人数： 学期：□春□秋</td><td colspan="2"></td><td></td><td></td><td colspan="2">□满意 □一般
□不满意 □希望更换</td></tr>
<tr><td>样书申请</td><td colspan="8"></td></tr>
<tr><td>已出版著作</td><td colspan="3"></td><td>已出版译作</td><td colspan="4"></td></tr>
<tr><td colspan="3">是否愿意从事翻译/著作工作 □是 □否</td><td>方向</td><td colspan="5"></td></tr>
<tr><td>意见和建议</td><td colspan="8"></td></tr>
</table>

填妥后请选择以下任何一种方式将此表返回：（如方便请赐名片）

地 址：北京市西城区百万庄南街1号 华章公司营销中心 邮编：100037

电 话：(010) 68353079 88378995 传真：(010)68995260

E-mail:hzedu@hzbook.com markerting@hzbook.com 图书详情可登录http://www.hzbook.com网站查询